

版权注意事项：

- 1、书籍版权归作者和出版社所有
- 2、本PDF仅限用于个人获取知识，进行私底下的知识交流
- 3、PDF获得者不得在互联网上以任何目的进行传播
- 4、如觉得书籍内容很赞，请购买正版实体书，支持作者
- 5、请于下载PDF后24小时内删除本PDF。

全栈应用开发

精益实践

黄峰达 著

全面介绍构建MVC应用全栈开发所需的完整知识体系
帮助读者以精益创业的思想开发Web应用

作者简介



黄峰达 (Phodal Huang)

程序开发者、创作者和作家，毕业于西安文理学院电子信息工程专业，现作为一名咨询师就职于 ThoughtWorks。他热爱编程、写作、设计、旅行、hacking，可从他的个人网站<https://www.phodal.com/>了解更多。

全栈应用开发

精益实践

黄峰达 著

电子工业出版社

Publishing House of Electronics Industry

北京·BEIJING

内 容 简 介

这不是一本深入前端、后台、运维、设计、分析等各个领域的书籍。本书以实践的方式，将一系列的领域及理论知识结合到一起，来帮助读者构建全栈 Web 开发的知识体系，并辅以精益及敏捷的思想，来一步步开发 Web 应用：从创建一个 UI 原型到编写出静态的前端页面；从静态的前端页面到带后台的应用，并部署应用；从 Web 后台开发 API 到开发移动 Web 应用。

在这个过程中，我们还将介绍一些相辅相成的步骤：使用构建系统来加速 Web 应用的开发；为应用添加数据分析工具来改进产品；使用分析工具来改善应用的性能；通过自动化部署来加快上线流程；从而帮助读者开发出一个真正可用的全栈 Web 应用。同时，我们也将帮助读者把这些步骤应用到现有的系统上，改进现有系统的开发流程。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。

版权所有，侵权必究。

图书在版编目 (CIP) 数据

全栈应用开发：精益实践 / 黄峰达著. —北京：电子工业出版社，2017.5

ISBN 978-7-121-31369-1

I. ①全… II. ①黄… III. ①网页制作工具—程序设计 IV. ①TP393.092.2

中国版本图书馆 CIP 数据核字 (2017) 第 078297 号

策划编辑：董 英

责任编辑：李利健

印 刷：三河市双峰印刷装订有限公司

装 订：三河市双峰印刷装订有限公司

出版发行：电子工业出版社

北京市海淀区万寿路 173 信箱 邮编 100036

开 本：787×980 1/16 印张：24.5 字数：441.2 千字

版 次：2017 年 5 月第 1 版

印 次：2017 年 5 月第 1 次印刷

印 数：3000 册 定价：79.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：(010) 88254888, 88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819 faq@phei.com.cn。

前言

学习 Web 开发最难的并不是学习相关技术，而是需要了解整个 Web 开发的知识体系。多数时候并不是因为我们不学习，而是因为我们不知道学习什么。完整的知识体系不仅仅包括前端、后台开发，还应该包括持续集成、自动化部署等内容。这些往往需要几本不同的书才能学习到，另外，它们也难以保证知识体系的完整性。我们在学习的时候，也往往并没有注意到它们之间的联系。

本书可以为读者构建出清晰、完整的 Web 开发体系，包括：前端、后台的技术选型，搭建构建系统，如何上线部署，并进行数据分析，以及如何在其中结合最好的工程实践等。

希望作为读者的你，可以将本书当作一本索引书籍，以此来开启你的 Web 开发新世界；你可以按书中的实践来进行 Web 编程，并结合理论来实践。

为什么写这本书

本书是我在实习的时候特别想写的一些内容——关于如何系统地学习 Web 开发，只是我一直缺少一条主线来将这些内容一一串起来。

2016 年年初，我在 GitHub 上开源了一个名为 Growth 的应用（读者可以在 App Store 和各大应用商店下载该软件）。在该应用中便包含了本书的主要思想：Web 应用的生命周

期。在不断迭代的过程中，该应用越来越受开发者喜爱，至今已经有超过 10000 名用户用过这个应用。随后，笔者在 GitHub 上推出了开源电子书《Growth：全栈增长工程师指南》，已经有超过 4500 个 Star。由于电子书本身只是一个指南，越来越多的读者还希望有一本实战。也因此诞生了《Growth：全栈增长工程师实战》，其在 GitHub 上也有超过 1000 个 Star。

后来，我才下决心去出版这样一本书。写一本书不是一件容易的事，相比较而言，读一本书则要简单许多。前者要花费一个人几个月的时间来完成，而后者只需要几星期、几天，或者是几小时的事。花几分钟将书的目录过一遍，随后只看几页想看的内容，余下的内容则可以在以后闲暇的日子里探索。

本书是我在编程生涯初期的一些体会，它更像是一本关于 Web 开发的索引书籍，但其实这些索引正是我读了大量书籍后，自己对精髓之处进行的理解加工。在这本书里，你会看到我对很多知识点进行了概括，并以实践的方式将一个个知识点连接到一起。

在最开始的时候，我曾经想将书名命名为“实习记”。后来又觉得虽然这是在我实习期间学到的知识，但其实很多内容在其他公司是学不到的。因此，在电子书里将其命名为 Growth，它不仅可以使读者增长知识，也在让我自己成长。

本书目标

本书的目标是帮助读者构建 Web 应用的全栈开发所需要的完整知识体系，并以精益创业的思想来一步步开发 Web 应用。

- 从创建一个 UI 原型到编写出静态的前端页面。
- 从静态的前端页面到后台的应用，并部署应用。
- 从 Web 后台开发 API 到开发移动 Web 应用。

在这个过程中，我们还将介绍一些相辅相成的步骤：

- 使用构建系统来加速 Web 应用的开发。

- 为应用数据分析工具改进产品。
- 使用分析工具改善应用的性能。
- 通过自动化部署加快上线流程。

从而帮助读者开发出一个真正可用的全栈 Web 应用。同时，我们也希望能帮助读者将这些步骤应用到现有的系统上，改进现有系统的开发流程。

本书结构

本书从结构上分成了 3 部分，每部分都会有不同的侧重点。

第 1 部分：准备阶段

在这一部分里，我们将主要集中于编码前的一系列开发准备工作，从选择一个合适的 IDE 到创建一个 Web 应用的构建流。

第 1 章 基础知识 介绍了搭建开发所需要的基本环境，以及 IDE、操作系统、版本管理工具等日常工具的选择与使用；还介绍了如何对一个目标进行切分，以便我们在实现的时候可以一步步往下实践。

第 2 章 最小可行化应用 介绍了如何使用 UI 工具来创建原型，并根据这个原型创建一个最简单的 Web 应用；接着介绍了在 Web 应用开发的过程中，如何使用精益的思想来开发出用户喜爱的产品。

第 3 章 技术选型与业务 对后台开发所需要的技术进行简单概览，并介绍了不同后台组件的框架，以及如何从这些框架中选择出合适的框架。同时还介绍了 Python 下的 Web 开发框架 Django，以及如何用这个框架创建一个“hello, world!”程序。

第 4 章 构建系统及其工作流 介绍 Web 应用中常见的构建流程及组件，以及如何结合 Fabric 打造后台的构建系统。

第 2 部分：编码到上线

在这一部分里，我们主要讲述大部分 Web 应用的开发过程，并介绍在开发过程中一些好的实践。

第 5 章 编码 介绍了如何使用 Django 创建一个简单的博客应用，以及如何使用单元测试、UI 测试来测试代码的功能。

第 6 章 上线 介绍了如何手动部署开发的 Web 应用到产品环境，以及如何使用自动部署工具来完成自动化部署。

第 7 章 数据分析和体验优化 介绍了如何使用网页监测工具来分析网页的流量来源、用户行为等，并结合一些前端、后台的优化工具对应用进行优化。

第 8 章 持续集成与持续交付 介绍了如何使用持续集成工具，以及如何使用持续集成工具来改进开发流程，并实现自动化的部署。

第 9 章 移动 Web 与混合应用 介绍如何编写后台 API 来创建移动应用，以及如何为单页面应用提供 SEO 支持。

第 3 部分：增量性优化

第 10 章 遗留代码与重构 介绍什么是遗留系统，以及如何基于第 2 部分中的经验来改进遗留系统。

第 11 章 增长与新架构 介绍如何使用回顾与反馈来使程序员成长，以及如何依据需要设计出新的架构。

技术栈概述

本书所介绍的工具主要集中于前端、后台、构建工具和前端 UI 框架四部分，分别如下。

- Django 是 Python 语言的一个 MVC 架构 Web 开发框架。本书使用这个框架来介

绍如何编写单元测试、功能测试，并演示如何使用它进行持续集成和持续部署。

- **Bootstrap** 是一个在前端领域相当流行的响应式 Web UI 开发框架，本书出于开发便捷的缘故使用这个框架。
- **Fabric** 是一个命令行的自动化部署工具，本书使用这个框架来展示如何搭建构建系统，并使用它来进行自动化部署。
- **Angular 2** 是一个可以用于构建移动应用和桌面 Web 应用的开发平台，我们在书里用它来展示如何开发前后端分离的 Web 应用程序。

上面的几个框架可以构成跨手机、桌面的一个 Web 应用，以及如何对其进行自动化部署。另外，还将介绍一些工具和框架来帮助我们开发：

- **Ionic 2** 是一个跨平台（Android、iOS、Windows Phone）的混合应用开发框架，基于 Angular 2 框架，并搭建有大量的 UI 组件，以及原生组件，我们在书里说明如何通过它与 Angular 2 共用代码来开发手机端应用。
- **Jenkins** 是一个持续集成工具，它提供了持续集成与持续部署工具链中所需要的大部分工具。我们将用它来展示如何进行持续集成，并结合 Fabric 来实现自动化部署。

本书将展示如何结合这些工具来做一些最佳实践，读者不必担心它会影响你的阅读，并且这些工具的替代品也很容易找到。

代码

本书相关的代码都可以从 GitHub 上下载到：<https://github.com/phodal/growth-code>。

混合应用部分的代码可以从 <https://github.com/phodal/growth-paper-hybrid> 处下载。

这些代码遵循 MIT 协议开源，读者可以将这些代码用于学习、商业等用途的项目中，不需要笔者授权。同时，笔者也不对这些代码的衍生代码负责。

遇到问题

在遇到问题时，欢迎及时与笔者联系。遇到代码问题时，建议直接在 GitHub 上创建一个相关 Issue，以便我们帮助其他读者解决同样的问题。

遇到内容不清楚等问题时，可以通过下面的方式联系笔者：

1. 通过 GitHub 上的 Growth 项目参与讨论：<https://github.com/phodal/growth-code>
2. 在 Growth 论坛上讨论：<https://forum.growth.ren/>
3. 在微博上与我联系：@phodal
4. 通过邮件：h@phodal.com
5. 加入 QQ 群讨论：529600394

你也可以在知乎、SegmentFault 网站上进行提问，并@phodal 来帮助你解决这个问题。

致谢

我要把这本书献给花仲马，没有她，就没有这本书。感谢她在这本书的写作过程中一直陪伴着我，并为这本书进行了中文校对来保证语句的通顺。

同时，我想特别感谢 ThoughtWorks 的同事薛倩、阿里巴巴的孙辉在本书创作过程中提供了详细的反馈，正是他们的帮助让本书更加准确、容易阅读。我还想特别感谢在 ThoughtWorks 学习时的同事，为我提供悉心指导与帮助。特别感谢王超、陈卿、王妮、曹隆凯、张静强、刘杰、王磊，在和他们进行结对编程时，我学习到了敏捷软件开发、Tasking 等编程之外的技能，感谢他们帮我走了这么远。

此外，还有那些在 GitHub 上为我提供反馈的用户，正是他们的反馈促使这本书更加完整。由于人数众多，这里仅列出这些用户的 ID：

感谢 ethan-funny、izhangzhihao、kaiguo、gymgle、aidewoode、wenzhixin、sasuke6、wangyufeng0615、walterlv、lolosssss、NehzUx、mikulely、yulongjun、PhilipTang、ReadmeCritic、ReadmeCritic、wangcongyi、loveisbug 等用户为《Growth: 全栈增长工程师指南》提供反馈与修改。

感谢 Pandoraemon、wo0d、ReadmeCritic、zhangmx、felixglow 等用户为《Growth: 全栈增长工程师实战》提供了反馈与修改。

轻松注册成为博文视点社区用户 (www.broadview.com.cn), 扫码直达本书页面。

- **提交勘误:** 您对书中内容的修改意见可在[提交勘误处](#)提交, 若被采纳, 将获赠博文视点社区积分 (在您购买电子书时, 积分可用来抵扣相应金额)。
- **交流互动:** 在页面下方[读者评论处](#)留下您的疑问或观点, 与我们和其他读者一同学习交流。

页面入口: <http://www.broadview.com.cn/31369>



目 录

第 0 章 绪论：Web 应用开发周期	1
0.1 Web 应用的生命周期	2
0.2 遗留系统与新架构	3
0.3 技术选型与验证	4
0.4 搭建构建系统	5
0.5 迭代	6
0.6 Web 应用开发步骤	7
0.7 小结	9

第 1 部分 准备阶段

第 1 章 基础知识	12
1.1 搭建开发环境	13
1.1.1 基本要素	13
1.1.2 常用效率工具及其在不同操作系统下的安装	14
1.1.3 搭建开发环境	22
1.1.4 开发工具	23

1.2	版本控制	27
1.2.1	Git 初入	28
1.2.2	Git 工作流	30
1.3	任务拆分	32
1.3.1	一本书的任务拆分	32
1.3.2	一个功能的任务拆分	33
1.4	小结	35
第 2 章	最小可行化应用	36
2.1	最小可行化产品	37
2.2	最小可行化 Web 应用	41
2.2.1	使用 Bootstrap 模板	41
2.2.2	完善原型	46
2.2.3	简单上线	47
2.3	精益与敏捷软件开发	52
2.3.1	敏捷软件开发	52
2.3.2	精益	56
2.4	小结	58
第 3 章	技术选型与业务	59
3.1	技术选型	61
3.1.1	后端选型	63
3.1.2	数据持久化	67
3.1.3	前端选型：UI 框架	71
3.2	Django	72
3.2.1	Django 简介	72
3.2.2	安装 Django	74
3.2.3	创建项目	77
3.3	从真实世界到代码	83
3.3.1	模型、领域、抽象	84

3.3.2 前后端分离	88
3.4 小结	90
第 4 章 构建系统及其工作流	92
4.1 构建流	93
4.1.1 搭建开发环境	96
4.1.2 准备生产环境	98
4.2 打造后端构建系统	100
4.2.1 使用 Fabric 搭构建系统	101
4.2.2 软件包管理	107
4.3 小结	109

第 2 部分 编码到上线

第 5 章 编码	112
5.1 创建首页应用	114
5.1.1 生成首页应用	115
5.1.2 编写第一个测试	122
5.1.3 使用 Selenium 进行功能测试	124
5.1.4 如何编写测试	128
5.2 创建博客应用	134
5.2.1 创建应用与博客管理	134
5.2.2 在页面上显示博客	141
5.3 数据与 Web 应用开发	150
5.3.1 管理数据	151
5.3.2 显示数据	151
5.4 小结	152
第 6 章 上线	155
6.1 手动部署	156

6.1.1	操作系统与服务器软件	157
6.1.2	第一次部署应用	162
6.1.3	配置管理	176
6.2	自动化部署	178
6.2.1	使用 Fabric 自动化部署	179
6.2.2	探索更优雅的方案	185
6.3	隔离与运行环境	187
6.4	小结	199
第 7 章	数据分析和性能优化	200
7.1	网站监测与分析	203
7.1.1	Google Analytics	203
7.1.2	自建监测和分析服务	212
7.2	性能分析及优化	214
7.2.1	前端优化：用 PageSpeed 工具分析和优化	215
7.2.2	后台优化：使用应用性能管理工具	223
7.2.3	使用 New Relic 进行优化	225
7.2.4	缓存初入	230
7.3	小结	234
第 8 章	持续集成与持续交付	236
8.1	持续集成与 Jenkins	237
8.1.1	工具选择与 Pipeline 设计	239
8.1.2	Jenkins 搭建持续集成	244
8.1.3	使用 Jenkinsfile 简化流程	252
8.2	持续交付与持续部署初探	255
8.2.1	持续交付	256
8.2.2	持续部署初探	260
8.3	小结	261

第 9 章 移动 Web 与混合应用	263
9.1 移动 Web 与单页面应用	264
9.1.1 单页面应用入门	266
9.1.2 API 设计与框架选型	272
9.2 创建移动应用	277
9.2.1 使用 Ionic 2 创建应用	278
9.2.2 更新首页	293
9.3 实现博客应用开发	297
9.3.1 创建博客 API	297
9.3.2 创建详情页和列表页	302
9.4 用户登录与博客创建	309
9.4.1 使用 JWT 实现登录	310
9.4.2 测试和发布应用	323
9.5 小结	325

第 3 部分 增量性优化

第 10 章 遗留代码与重构	328
10.1 遗留系统	330
10.1.1 什么是遗留系统	330
10.1.2 遗留系统改造	334
10.2 易读的代码与重构	336
10.2.1 命名	337
10.2.2 一次只做一件事	339
10.2.3 减少重复代码	340
10.2.3 排版	342
10.2.4 重构	343
10.3 小结	346

第 11 章 增长与新架构	348
11.1 增长	350
11.1.1 增长：回顾与改变	350
11.1.2 增长：技能学习与构建索引	354
11.2 设计新架构	357
11.3 小结	363

附 录

附录 A 如何学习新的技术	366
附录 B 安装 Piwik	372

第 0 章

绪论：Web 应用开发周期

这部分内容最早出自笔者写的文章《RePractise: Web 开发的七天里》，原文简单描述了 Web 应用的生命周期。后来发现，这条路几乎是所有 Web 应用的必经之路。一个 Web 应用在其生命周期里，都要经历搭建开发环境、创建构建系统、编写代码、进行数据分析等，直至最后使用新的系统来替换这个遗留系统。

作为本书的开头，笔者难免想说些废话，初学者可以跳过这一部分。等到阅读完本书再看看这部分内容，或者等完全经历了一个项目的开发过程，再回过头来看这部分的内容就会有所体会。如果你是一个有经验的开发者，相信你对这个生命周期一定也深有体会。

0.1 Web 应用的生命周期

在我所经历的项目以及我所看到的 Web 应用里，它们都有相同的很有意思的生命周期。我们经常在网上看到某个知名的网站使用某个新的技术、语言来替换旧的系统，某个 App 使用新的框架来替换现有的 App。我们所看到的都只是这些公司正在重构现有的系统，这实际上是一个周期的结束，以及一个新周期开始。其过程如图 0-1 所示。



图 0-1 Web 应用的生命周期

仔细一想就会发现：我们所经历的项目都在以不同的时间长度经历相同的生命周期。

0.2 遗留系统与新架构

在我开始工作的时候,接触的第一个项目就是一个遗留系统。在一次休息时,我们在比赛找最古老的源码文件,最后找到了10年前写下的一个文件。尽管在我们的代码里有单元测试、针对具体业务功能的测试,项目的代码已经超过20万行,项目中仍然有相当多的代码超出了我们所理解的业务范围。毕竟在这些年头里,有相当多的功能已经不存在了。后来,我们选用微服务重构了这个系统。对于中型的遗留系统来说,这算是一剂良药。

让我们先从某某网站使用新架构重新设计说起。当我们决定使用新架构重新设计系统时,原因可能是多种多样的,如果我们排除一些无法抗拒的因素(如政治),那么剩下的原因可能就只有两个。

- 系统已经变得难以维护。这里的原因仍然有很多:大量的代码已经没有人知道其业务逻辑,变得难以修改;代码间耦合度过高,重构系统的难度过于复杂;项目所使用的技术栈已经过时,已经被市场所淘汰;团队的技术栈在成员变动的过程中,团队中大部分成员的技术栈已经和当前的项目不匹配了。
- 系统的技术栈已经难以符合业务的需求。绝大多数情况下,我们在最初开始创建项目的时候,所选择的技术栈都是符合当时业务需求的技术栈、可以快速验证其业务价值的技术栈。而随着业务的扩张,现有的技术栈很快将难以满足当前业务的需求,或出现性能优化上的限制。

在多数情况下,我们都会将这种系统称为遗留系统。在这时团队里的气氛便是“能不动这些代码就尽量不去动它”。我们已经很难将项目的问题归为人的因素,多数时候都是受业务扩张的影响。作为一个专业的程序员,我们的本能就是将程序写好,而我们往往没有这样的机会。

业务人员对项目经理说:“我们的竞争对手已经在本周上线了这个功能。”

项目经理对开发人员说：“这个功能下星期就要上线！”

是的，我们的功能不得不马上上线。这时候，我们往往要在代码质量和交付速度上做出一些妥协。妥协多了，系统也就变烂了。

开发人员说：“这个代码我不太敢修改，要是出了什么大 Bug 怎么办？”慢慢地人们就开始讨论起重构系统的事宜，并开始着手设计新的架构——使之可以满足当前的业务需求、可预测时间内的业务与技术需求。

0.3 技术选型与验证

在讨论新架构的过程中，不同的人可能会有不同的技术偏好，也会因存在一些政治因素导致不同技术方案的产生。如团队中的一些人可能出于稳定缘故而选择 Java，一些人可能出于对新技术的需求选择 Scala，而另外一些人可能考虑到团队中大部分人可能因为都会使用 JavaScript 而选择使用 JavaScript。如图 0-2 所示，我们的考虑应该不仅仅取决于这一系列的技术因素。

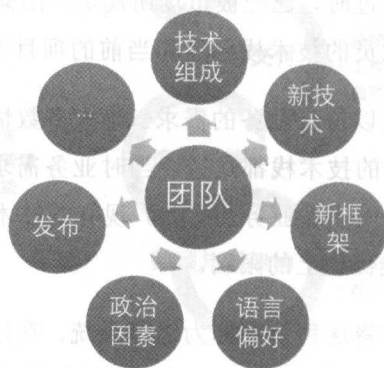


图 0-2 技术选型考虑因素

需要注意的是：在做技术选型的时候，要尽最大可能以团队为核心。在做决定之前，我们要提出不同语言、框架下的技术模型，并且进行验证。随后就需要快速搭建出一个原型，并针对这个原型进行假想式开发，然后验证原型本身是经得起考验的。

在这一阶段，我通常喜欢在 GitHub 上搜索一些名字中带有 boilerplate 的项目，即模块文件。而当一个框架很流行的时候，我就会去相应的 awesome-xx 寻找，如 awesome-react 就可以寻找到 react 相关的项目集。然后，克隆这样一个项目，开始依照现有的系统创建简单的 Demo。随后，就可以依据我们的业务试着在这上面进行扩展。最后，再决定是否使用这门技术和这个框架。

通常来说，在选择一门新技术设计系统时，需要承担的风险相当大，而如果能成功，那么它可能会带来巨大的收益。从这点看，使用最新的技术与赌博无异。在一些成熟的公司里，会有专门的技术委员会负责对新技术进行审核，来决定是否可以在某个项目里使用新技术。除了考虑其为开发带来的便利性，他们更多地还会考虑其成熟度、安全和技术风险等。

0.4 搭建构建系统

决定好架构并选择完技术栈后，我们就开始着手创建项目的构建系统，设计项目的部署流程。构建系统不仅包含项目相关的构建流程，还从某种意义上反映了这个项目的工作流程。

创建完“hello, world”程序后，我们要着手做的事情就是创建一个持续集成环境。这样的环境包含一系列的工、步骤及实践，从工具上说，我们需要选择版本管理工具、代码托管环境、持续集成工具、打包工具、自动部署脚本等一系列流程，这些流程将会在第4章详细讨论。

图 0-3 便是笔者之前经历过的一个项目的构建流程。

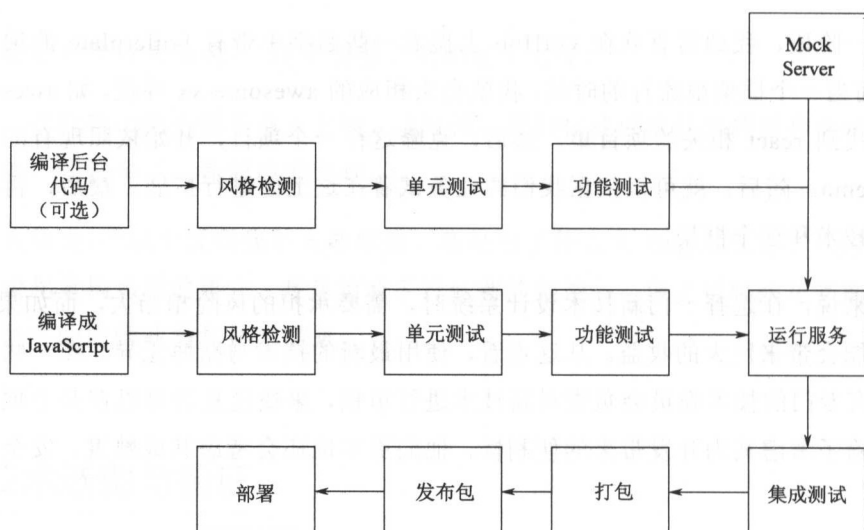


图 0-3 构建过程

这是一个后台语言用 Java、前台语言用 JavaScript 的项目的构建流程。

0.5 迭代

在互联网行业里，能越快速地对市场需求做出反应，就越能有更好的发展。只要你细心观察就可以发现，大部分互联网公司都在以一定的规律更新产品，或者一周，或者两周，又或者一个月等，这种不断根据反馈来改进产品的过程称为迭代。如图 0-4 所示是一个简化的迭代模型。

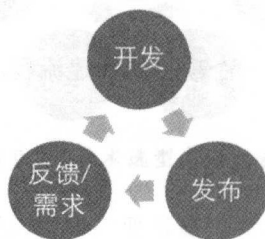


图 0-4 简化的迭代模型

当一个迭代开始时,我们需要收集上一个迭代的反馈或者新的需求,然后开始开发代码,最后再发布产品。开发的产品在这个过程中不断地增强功能。为此,还需要选择一个好的迭代周期。一个好的迭代周期既应该有充足的时间修复上一个迭代的 Bug,又能在下一个迭代开始之前交付重要的功能。当然,如果交付的软件包里出现了重要的 Bug,那么我们也能在第一时间使用旧版本的包,并在下一个迭代交付。在这样的开发节奏里,一周显得太短,一个月又显得太长,两周会是一个很不错的时间。

当一个团队在这方面做得不好时,那么他们可能在一次上线后,发现重要的 Bug,不得不在当晚或者第二天更新他们的产品。即使是有经验的团队,在开发初期也会经常遇到这些问题,而这些问题可以依赖于在迭代中改进。好的迭代实践都是依据团队自身的需求而发展的,这意味着有时候适合团队 A 的实践并不一定适合团队 B。

随后,我们会在这个“hello, world”的基础上不断添加各种功能。

0.6 Web 应用开发步骤

令人难以置信的是,我们做了这么多事情以后还没有开始写代码。事实上,在这一步里,我们已经搭建好了一个最小可运行的 Web 应用。在这之后,我们所要做的事情就是提交代码即可。将代码从本地提交到服务器后,持续集成服务器将帮我们运行测试,在测试通过后,打包、发布现有的代码,最后部署到测试环境里。

(1) 编码

如果不考虑技术难度的因素,写代码看上去就是一件很简单的事。我们只需要按照需求,将功能一点点往上叠加即可。如果不考虑这个过程中添加的代码质量,将会得到一个难以维护的系统,并且在拿到需求后的第一反应也并非直接开始实现功能,而是首先应该考虑可以将这个需求拆分为几步,我们将这个过程称为 Tasking。

假如,我们正在实现某个详情页的显示功能,它依赖于前端和后台。那么可以直接先做后台 API,再实现前台 API,最后依据需要微微调整 API。我们也可以先用 Mock 的数

据实现前端页面，再依据定义出来的数据格式实现后台 API。在这两种不同的实现中，我们都有一个明确的先后步骤。同样，对于一个更加复杂的功能来说，需要切分得更加仔细，每一次只挑选其中一个任务，实现后，再一步步往下执行，最后实现这个功能。

有意思的是，当我们已经决定切分为多步来实现功能的时候，就可以在每一步里进行几次不同的代码提交，以便以后知道每一步中做了什么内容。如果只是在最后一步直接提交代码，那么在未来修改代码时，便难以理清当时的思路。

一个合理的编码过程不仅包括功能的实现，还应该有测试。尽管出于项目进度的原因，多数项目都不存在测试，而正是因为没有测试，使得整个项目更加混乱。新的功能容易影响旧有的代码，除非有足够多的测试人员，否则我们无法保证所有的功能都是正常的。在有限的条件下，我们应该编写重要的测试，以保证核心功能不被破坏。在条件允许的情况下，我们应该尽可能地保证测试对重要功能的覆盖。由于代码库不只有一个人在提交，如果在某次提交中测试被破坏了，就可以知道谁破坏了测试，他/她应该有责任来修复这个测试。

在完成功能后，我们还可以对代码进行重构，以此来保证代码的质量。此外，在日常工作中，我们会用 Code Diff（代码检视）来帮助大家提高代码质量。因此，并不是实现了功能就完事了，我们应该尽量保证代码的质量。

（2）上线和数据分析

好了，现在是时候上线了。在以前，上线就是登录到服务器做数据备份。随后，在本地构建、上传软件包，安装软件的依赖。最后，重启服务器、Done。

在今天看来，这是一件相当费力的事，我们可以使用自动部署工具来加快这个流程，甚至当我们有足够的测试覆盖率时，可以直接将测试通过的代码直接部署到产品环境。不过，要这样做应有相当的技术能力，并且要保证我们可以协调好开发人员、运维人员等。从技术上说，这可能是一件容易的事，但是从组织结构上说，这并不是一件轻松的事。

而故事并没有因此而止步于上线，在产品上线时，我们可以通过数据分析工具来监测用户的行为、网站的访问量等信息。

对开发人员来说，这样的分析平台可以帮助我们解决用户在使用过程中遇到的 Bug——他在哪一步出的问题？他在出问题前做了什么操作？

对业务人员来说，他们可以借此来分析产品受欢迎的程度、用户及流量来源、转化率等信息，并依此来对着陆页、转化率等进行优化。几种常见的流量来源包括搜索引擎、外部链接、付费搜索等，这些都可以依此来做出一些调整。从技术角度说，我们可以提高网站的 SEO（搜索引擎优化）水平来添加流量，这将在第 7 章中进行讨论。

0.7 小结

本章我们对本书的内容进行了一个简单概述，并完整地介绍了它们之间的联系。同时还介绍了在阅读过程中，我们将学习到的内容，以及将遇到的一些挑战。

xxx: Hi Phodal, 欢迎来到 Growth Studio 项目组。我是 xxx, 今天由我来带领你进入第一篇, 我们在进入一个项目之前, 需要做什么?

Phodal: 先进行调研。

xxx: 什么意思, 在我们运行某个业务的代码之前需要做什么?

Phodal: 安装语言环境, 还是安装 IDE?

第 1 部分 准备阶段

在这一部分里, 我们将主要精力集中于“项目开始前”的一些准备工作, 如搭建基础的构建系统、从业务角度对技术进行选型等。同时, 我们还将关注一些特别有意思的东西, 如 Web 应用的生命周期、对不同业务的技术栈考虑等。

1.1.1 基本要素

一般来说, 在一个新的计算机上搭建开发环境时, 总会先安装好下面几个工具。

- 包管理工具。包管理工具允许用户用命令行或者 UI 界面的方式直接安装软件。例如, 如果我们想在 Ubuntu 上安装 Vim 编辑器时, 可以直接在命令行输入如 `sudo apt install vim`, 就可以安装 Vim 编辑器。并且这样的工具存在于不同的操作系统 (Windows、MacOS、类 UNIX 系统) 上。对 MacOS 和 Windows 系统需要独立安

第 1 章

基础知识

在本章中，我们将带领读者搭建好基本的开发环境——在不同的操作系统中，如 Windows、Mac OS、Linux，以及如何选择 IDE 和 Editor 的一些偏好及设置，并引入版本管理系统及 Git 等基础软件工程的知识。

xxx: Hi Phodal, 欢迎加入 Growth Studio 项目组。我是 xxx, 今天由我来带你进入第一篇。我们在进入一个项目之后, 需要做什么?

Phodal: 先运行代码。

xxx: 你再想想, 在我们运行某个语言的代码之前需要什么?

Phodal: 安装语言环境, 还是安装 IDE?

xxx: 等等, 让我们先按顺序来, 为了克隆代码, 需要使用版本管理工具; 为了修改代码, 需要一个编辑器; 为了运行代码, 需要安装语言本身的环境。那么, 你觉得我们先从哪一步开始呢?

Phodal: 都可以。

现在, 让我们先了解并搭建开发环境吧。

1.1 搭建开发环境

搭建好顺手的开发环境并不是立马就能做到的事, 不同的开发者有不同的偏好、技术信仰等。这些因素会导致每个人构建完的开发环境有所差距。尽管如此, 总的来说, 这些工具都会有一些相似的特征。同时, 它们还需要有一些基本要素。

1.1.1 基本要素

一般来说, 在一个新的计算机上搭建开发环境时, 总会先安装好下面几个工具。

- 包管理工具。包管理工具允许用户用命令行或者 UI 界面的方式直接安装软件。例如, 当我们想在 Ubuntu 上安装 Vim 编辑器时, 可以直接在命令行输入如 `sudo apt install vim`, 即可以安装 Vim 编辑器, 并且这样的工具存在于不同的操作系统 (Windows、MacOS、类 UNIX 系统) 上。对 MacOS 和 Windows 来说需要独立安

装额外的软件。

- 命令行环境。如果你已经使用了命令行来代替日常的操作，则可能还需要一个好的命令行运行工具。如果你使用的是 Windows，则需要有其他的命令行辅助工具来加速你的开发。
- 编辑语言环境。多数情况下，我们的日用操作系统上都没有相应的语言运行环境，需要安装对应语言的安装包，及其对应的包管理环境等。
- IDE 或者编辑器。在编写代码时，需要一个 IDE 或者编辑器来编辑代码。
- 一个或多个现代的浏览器。在开发前端用户界面时，只会用某个特定的浏览器来开发，这时容易因为我们使用了一些特定内核（如 Webkit）的浏览器，而导致界面在其他浏览器运行时出现布局问题。因此，至少需要一个额外的浏览器来保证运行效果，Google Chrome 和 Firefox 在这方面都是一个不错的选择，它们有相当丰富的扩展插件可以使用。建议读者不要使用 IE 10 以下的浏览器来充当开发用的浏览器——除非需要兼容 IE 10 以下的浏览器。
- 数据库软件。同样，为了使用数据库，需要使用数据库软件。在开发时，SQLite 和 MySQL 都是不错的选择。

在这些工具中，有很多是可以直接通过包管理工具来安装的。因此，先让我们了解一下包管理的概念及其作用。

1.1.2 常用效率工具及其在不同操作系统下的安装

在日常使用计算机时，可以安装下面的工具来提高效率。

- 包管理工具。
- 命令行环境。
- 快速启动工具。

下面让我们来逐一了解一下这些工具。

1. 包管理工具

包管理不仅存在于操作系统中，还存在于不同语言的环境里。如前面所说在操作系统中安装软件，最方便的东西莫过于包管理了。引自 OpenSUSE 官网的说明图如图 1-1 所示。

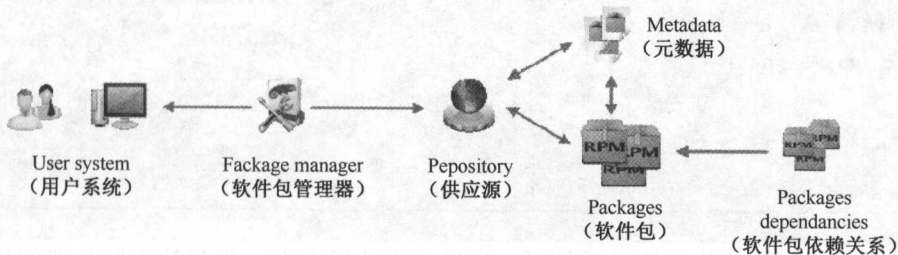


图 1-1 包管理

在图 1-1 中包含几个基本的元素。

- **软件包 (Packages):** 软件包不止是一个文件，内含构成软件的所有文件，包括程序本身、共享库、开发包以及使用说明等。
- **元数据 (Metadata):** 包含于软件包中，包含软件正常运行所需要的一些信息。软件包安装之后，其元数据就存储于本地的软件包数据库中，以用于软件包检索。
- **软件包依赖关系 (Packages dependencies):** 它是软件包管理的一个重要方面。实际上，每个软件包都会涉及其他的软件包，软件包里程序的运行需要有一个可执行的环境（要求有其他的程序、库等），软件包依赖关系正是用来描述这种关系的。

当我们使用某个命令去安装软件时，将会运行一系列的操作才能完成安装。以在 Ubuntu 上安装 VIM 为例：

```
phodal@ubuntu:~$ sudo apt install vim
```

```
正在读取软件包列表... 完成
```

```
正在分析软件包的依赖关系树
```

```
正在读取状态信息... 完成
```


将会同时安装下列软件：

```
vim-runtime
```

建议安装：

```
ctags vim-doc vim-scripts vim-gnome-py2 | vim-gtk-py2 | vim-gtk3-py2 |  
vim-athena-py2  
| vim-nox-py2
```

下列【新】软件包将被安装：

```
vim vim-runtime
```

升级了 0 个软件包，新安装了 2 个软件包，要卸载 0 个软件包，有 0 个软件包未被升级。

需要下载 6,247 kB 的归档。

解压缩后会消耗 30.2 MB 的额外空间。

一般来说，在初始化包管理系统后，我们在本地缓存有一份包管理服务器的软件包列表。每次安装某个软件时，将会读取这个软件包清单，查找其中是否包含这个软件包。随后，开发构建、查找出这个软件所依赖的软件包，并保证这些包是可用的。最后再安装软件包的依赖及该软件包。

同样，对语言的包管理工具来说也是如此。以 Node.js 为例，我们需要检查服务器上是否有对应的软件包，下载它的 package.json 文件后，再安装其 package.json 依赖的软件包，安装依赖后，再安装这个依赖包。

作为一个专业的程序员，我们经常会使用各式各样的包管理工具来加速日常使用。而不只是简单地通过 Google 搜索某个软件，然后下载并安装。

2. 命令行环境

我们已经在上面对包管理工具进行了简单介绍，与包管理工具不一样的是，命令行环境存在于不同的系统中。只是并不是所有的操作系统的命令行工具都会让你觉得顺手。多数时候，我们使用额外的工具来使系统更加方便使用。如在类 UNIX 系统里，我们使用 Zsh¹来替换默认的 Bash 作为终端软件。Zsh 在兼容 Bash 的同时还提供了更好的自动补

¹ Zsh 是一款功能强大的终端（Shell）软件，既可以作为一个交互式终端，也可以作为一个脚本解释器。

全、更好的文件名展开等改进。配置上 Oh My Zsh, 就可以做出一个兼具实用与美观的终端, 如图 1-2 所示。

```
fdhuang@PHODAL growth-paper master X* ls
css
ebook.md
epub.css
images
img
index.html
listings-setup.tex
resources
style.css
template
图灵图书选题表 (作者).doc

fdhuang@PHODAL growth-paper master X* git st
On branch master
Your branch is up-to-date with 'origin/master'.
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   chapters/00-introduction.md
    modified:   chapters/01-basic.md

no changes added to commit (use "git add" and/or "git commit -a")

fdhuang@PHODAL growth-paper master X*
```

图 1-2 Zsh 示例

在图 1-2 上半部分, 终端使用了不同的颜色和状态来表示不同类型的文件, “growth-paper” 部分用于显示文件夹, 白色字体用于显示普通的文件。在图 1-2 下半部分中, 可以看到下面的部分里显示 Git 的状态, 即当前有修改, 如图 1-3 所示。

```
fdhuang@PHODAL growth-paper master X*
```

图 1-3 Git 状态

当我们执行 `git status` 时, 它用黄色标识出对这些文件进行的修改。另外, 它还能用不同的颜色标识出不同文件的状态。顺便提一句: 对 Git 来说, 使用如 SourceTree 之类的图形界面也是一个不错的选择。

3. 快速启动工具

除了上面的两个工具, 我们还应该有一个快速启动工具。直接使用快捷键来打开这个工具, 输入想要打开的程序的名字, 再按下回车键, 应该可以直接运行这个程序, 或者打开某个文件。如图 1-4 所示的是笔者日常使用的快速启动工具 Alfred。

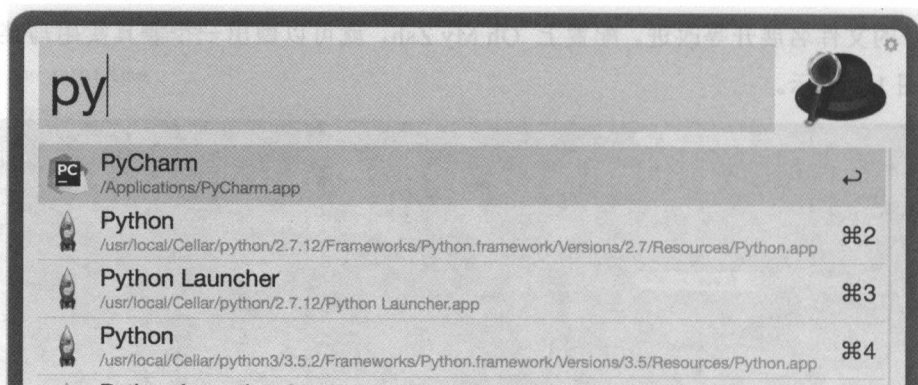


图 1-4 Alfred 截图

输入 `py` 时，它会寻找系统中对应的程序，我们只需要按下回车键即可。

现在，我们来看看在不同的操作系统上如何安装这些工具。

4. 在 Windows 上安装

(1) 包管理工具：Chocolatey

在 Windows 操作系统中，我们可以使用 Chocolatey (网官地址: <https://chocolatey.org/>) 作为包管理工具。它的安装过程也相当简单，只需要用管理员权限运行命令提示符，并粘贴下面的代码即可：

```
@powershell -NoProfile -ExecutionPolicy Bypass -Command "iex ((New-Object System.Net.WebClient).DownloadString('https://chocolatey.org/install.ps1'))" && SET "PATH=%PATH%;%ALLUSERSPROFILE%\chocolatey\bin"
```

当然也可以直接运行 PowerShell，然后输入：

```
iex ((New-Object System.Net.WebClient).DownloadString('https://chocolatey.org/install.ps1'))
```

建议读者访问 <https://chocolatey.org/install> 获取安装脚本来安装。

注意：你需要 Windows 7 以上的操作系统，或者 Windows Server 2003 以上的系统，

并且需要有 .NET Framework 4 以上, 以及 PowerShell v2 以上。

安装完成后, 就可以直接运行 `choco install + 软件名称` 来安装对应的软件, 例如, 将在下面介绍的 Wox (见图 1-5)。

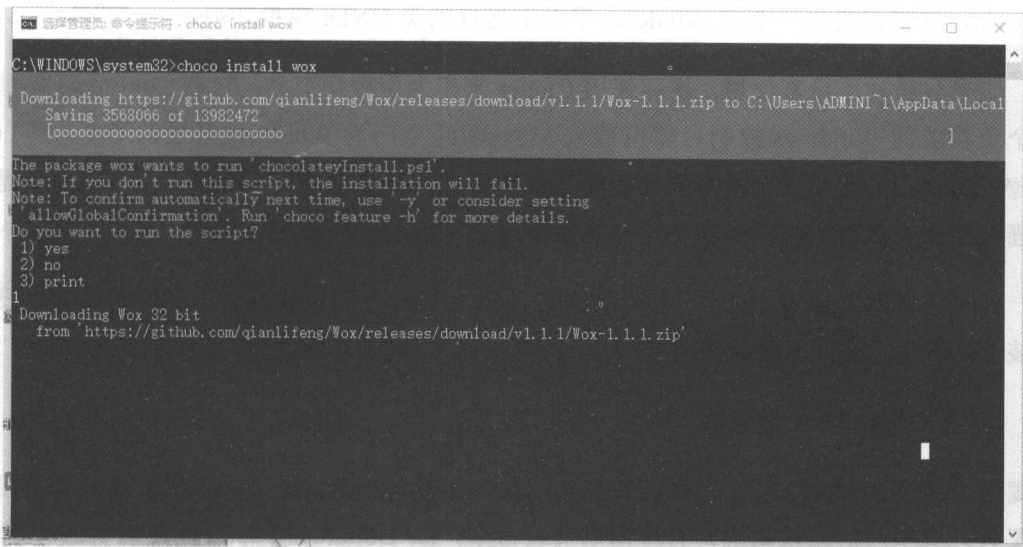


图 1-5 Choco 安装 Wox

在这个过程中, `choco` 将从服务器获取对应软件包的下载地址, 并下载对应的软件包, 执行相应的安装脚本来安装软件。

注意: 因为 `choco` 是直接从相应软件指向的服务器上下载软件的, 所以在一些网站上 (如 GitHub) 下载时, 可能因为网络问题而无法访问或者下载中断。

(2) 快速启动工具: Wox

Wox 就是在前面提到的快速启动工具, 除了上面提到的功能, 它还可以打开百度、Google 进行搜索, 甚至通过一些插件的功能实现单词翻译、关闭屏幕等更多的功能。另外, 它还能支持中文拼音的模糊匹配。

你可以直接从官网获取该软件, 官网地址为: <http://www.getwox.com/>, 并在安装完 Python 语言环境后, 即可使用 Wox 来快速启动工具。

(3) 命令行工具：cmdr

Windows 自带的 CMD 缺乏一系列的功能，难以完成日常的开发工作。当我们习惯使用类 UNIX 上的命令行工具，或者准备往类 UNIX 操作系统上迁移时，可以考虑使用 Cygwin，它可以提供一个在 Windows 平台上运行的类 UNIX 模拟环境。

如果只是为了日常使用，推荐使用 Cmder 来作为命令行工具。它把 ConEmu（提供了一个全功能的 Windows 控制台模拟器）、MSysGit [专门为 Windows 开发的 Git 工具（Git For Windows），它集成了所需要的运行环境和组件，直接安装即可使用] 和 Clink（提供了强大的命令行工具）打包在一起，它附带了漂亮的 monokai 配色主题。

你可以访问 Cmder 的官网（<http://cmder.net/>）自行下载该软件。如果你的电脑上没有安装 Git 环境，应该下载全功能（Full）版本，而不是 Mini 版。

5. 在 GNU/Linux 上安装

对于 GNU/Linux 系统，如 Ubuntu、CentOS、OpenSuSE 等来说，它们都自带了包管理工具。对不同的系统来说，有不同的安装命令，如 Ubuntu 的 apt（在低版本的 Ubuntu 系统中使用 apt-get）、CentOS 的 yum，以及 OpenSuSE 的 yast 等。因此，建议读者参照自己所用的操作系统来使用。

(1) 命令行工具：Zsh

同样，对类 UNIX 用户来说，建议读者使用 Zsh，并且可以搭建 Oh My Zsh 使用。Zsh 可以直接使用包管理工具来安装。

Oh My Zsh 是一个开源的、社区驱动的 zsh 配置管理框架。它托管在 GitHub 上，有超过 1000 个开发者为这个项目做出贡献，并且它有 200 多个插件，以及多达 140 个主题。安装方法相当简单。

我们可以通过 curl 命令来安装：

```
sh -c "$(curl -fsSL https://raw.githubusercontent.com/robbyrussell/oh-my-zsh/master/tools/install.sh)"
```

或者用 `wget` 命令安装：

```
sh -c "$(wget https://raw.githubusercontent.com/robbyrussell/oh-my-zsh/  
master/tools/install.sh -O -)"
```

注意：在很多操作系统上，`curl` 和 `wget` 都需要单独安装。

（2）快速启动工具：Launchy

同样，我们也需要在 GNU/Linux 操作系统上安装一个快速启动工具，Launchy 就是一个不错的选择。它是一个跨平台的快速启动工具，只需要按下 `Alt+空格键` 就可以打开它的主界面，然后就可以和 Alfred 一样输入名字调出应用程序。

6. 在 Mac OS 上安装

（1）包管理工具：Homebrew

与 Windows 操作系统相似的是，Mac OS 上也没有自带的包管理工具，因此，需要通过第三方软件来实现包管理功能。在 Mac OS 下有两个不错的包管理工具，一个是 MacPorts，另一个是 Homebrew，前者因为权限和依赖问题而饱受诟病。其安装方法也相当简单，只需要执行下面的命令（也可以直接访问 <http://brew.sh/> 来获取）即可：

```
/usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/  
install/master/install)"
```

Homebrew 依赖于 Xcode Command Line Tools 来编译和下载软件。因此，如果没有安装该软件，在安装过程中会提示安装该软件。如果使用该工具来安装 `wget`，其命令如下：

```
brew install wget
```

Homebrew Cask 是 Homebrew 的扩展，它可以基于 Homebrew 来安装带有图形界面的应用，而不是通过拖拉来安装程序。例如，无法通过 Homebrew 来安装 Google Chrome 浏览器，但可以通过 Homebrew Cask 来安装它。其安装方式如下：

```
brew tap caskroom/cask
```

随后，可以通过下面的命令来安装 Google Chrome 浏览器：

```
brew cask install google-chrome
```

(2) 命令行工具：iTerm 2

Mac OS 自带的终端仅提供了一些基本的终端功能，而 iTerm 2 在这上面提供了更多的功能，如切割窗格、热键切换窗口、搜索、自动完成等功能，并且只需要配合好上面的 Zsh 和 Oh My Zsh 就可以制作出一个漂亮的控制台。

1.1.3 搭建开发环境

除了上面提到的这些工具，还需要安装 Python 语言的运行环境，由于我们将在书中使用 Django 作为 Web 开发框架，其官方文档中不同的 Django 版本对 Python 有不同的版本要求，见表 1-1 所示。

表 1-1

Django 版本	Python 版本
1.8	2.7、3.2（至 2016 年年底）、3.3、3.4、3.5
1.9、1.10	2.7、3.4、3.5
1.11	2.7、3.4、3.5、3.6
2.0	3.5+

考虑到 Django 2.0 后的版本只支持使用 Python 3.5 及以上版本，这里将使用 Python 3.5 来开发 Web 应用。

对 Windows 用户来说，我们需要从 <https://www.python.org/> 处下载 Python 3.5，或者直接使用 choco 来安装。在 Ubuntu 16.10 版本的系统上，它自带了最新版本的 Python。在 Mac OS 系统上，可以通过上面介绍的 brew 来安装。

另外，你还需要在计算机上安装一个 Chrome 浏览器，并安装一个 Git 客户端来进行版本管理。

如果不习惯使用命令行进行版本管理，可以考虑试试 SourceTree。它是一个免费的 Git 客户端，并且提供一个扁平化的可视界面，如图 1-6 所示。

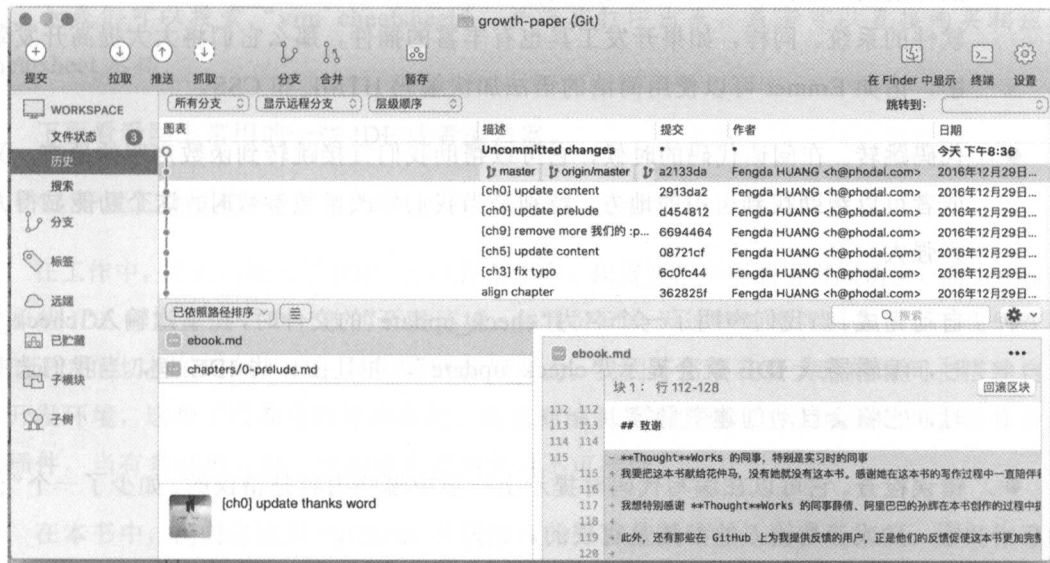


图 1-6 SourceTree 截图

我们可以直观地查看分支情况、提交信息、修改内容，并且可以进行拉取、提交代码等操作。

1.1.4 开发工具

如果你是一名编程新手，建议你使用 IDE。随后，可以考虑自定义自己的编辑器。

在编辑器里无论是 Vim 还是 Emacs，或者是流行的 Visual Code 或者 Atom，都需要花费时间去配置，以使编辑器可以像 IDE 一样强大。但是怎样才算和 IDE 一样强大呢？你需要有下面的一些东西。

- **语法高亮。**这是最基础的功能（如果你正在使用的编辑器，如记事本，不支持该功能，请不要再使用这样的工具。），它能依据语言中的关键词使用不同的颜色和字体区分代码。它可以增强人对编辑器中内容的可读性，来降低误读和误写

的概率。

- **插件功能。**选择手机操作系统与选择计算机操作系统一样，我们会选择具有丰富软件的系统。同样，如果开发工具也有丰富的插件，那么它们将大大提高开发效率，诸如 Emmet 可以使用简洁的语法加快编码 HTML 和 CSS。
- **代码跳转。**在阅读代码的时候，它可以帮助我们直接跳转到函数声明的地方，又或者可以帮助找到引用的地方。特别是当我们修改函数参数时，这个功能显得尤为强大。
- **自动完成。**当我们声明了一个名为“check_update”的文件时，在下次输入“check_”时，编辑器或 IDE 就会提示“check_update”，并且在一些 IDE 里，当我们选中时，它将会自动创建变量等。
- **错误检查。**它可以在编写代码时提示上一步中哪些内容是错误的，如少了一个“;”号，又或者是输入的参数是有误的。
- **版本控制集成。**使用版本控制软件是编程人员的基本技能，好的编辑器会集成这样的功能，我们可以直接在编辑器里比较代码的修改，查看上次的修改时间等，又或者直接提交代码。
- **依赖管理。**当我们自己去写 import xx 语句来引用某个库的时候，还需要花费时间去找到对应的类。当开发工具上有这样的功能时，只需要先编写代码，再调用快捷键导入依赖即可。
- **调试。**对前端开发人员来说，浏览器就是最好的调试工具。对后台人员来说，只有“print”是不够的，还需要深入每一步函数调用，才会发现问题出在哪里。
- **高级功能。**如重构等。笔者一直很喜欢 JetBrains 系列的 IDE，是因为它对重构有很好的支持。

无论是 Emacs 和 Vim 这样的编辑器，还是 WebStorm 这样的 IDE，我们都需要花时间去掌握它们的快捷键。

提示：当我们需要某个语言、框架的常用功能、函数等，又或者是编辑器的快捷键时，可以直接搜索其对应的 cheatsheet（即小抄），通常会在一页 PDF 或者图片里显示其全部功能。如我们可以搜索“vim cheatsheet”，并将其打印出来，或者可以直接购买相应的 cheatsheet 水杯。

下面看看网上常用的一些 IDE 或者编辑器。

1. 开发工具推荐

在工作中，笔者习惯使用 IDE。在项目工作中，我需要在不同的语言间切换，而 JetBrains 的那些 IDE 可以提供一个一致的开发环境，并且公司会为这些效率工具买单。当我们在不同的语言间切换的时候，IDE 显得特别有战斗力，我们不需要花费大量的时间去搭建自己的开发环境，这些工具都可以开启即用。而选择编辑器则需要花费大量的时间去选择合适的插件，当有多种语言时，这种成本开始变得不可接受。

在本书中，我们将使用 PyCharm 社区版作为开发工具，它是免费版本的。如果你寻求更强大的功能，则可以考虑使用专业版。

除了 PyCharm，我们也有一些不错的编辑器可以使用。

- **Sublime Text**：是一款相当流行的收费图形编辑器。由于其本身使用了 Python 语言作为开发语言，其对 Python 语言的支持相当好。它带有代码缩略图、完整的 Python API、Goto 功能等。它还带有一个非常丰富的插件系统，前提是要手动安装社区驱动的包管理器：**Package Control**。它可以同时支持 Windows、Linux、Mac OS X 等操作系统。
- **Atom** 和 **Visual Code**：两者都是开源且免费的编辑器，都有相当丰富的插件。如果你的电脑配置不错，可以考虑使用 Atom 或者 Visual Code 作为编辑器，由于其基于 Electron（基于 Chromium 浏览器），因此，需要消耗更多的内存。值得一提的是，Atom 提供了更丰富的插件系统，而 Visual Code 可以提供更流畅的体验。
- **Vim**：其和 Emacs 是在 Linux 和 UNIX 系统中最常用的两种文本编辑器。它是从 vi 发展出来的一个文本编辑器，它遵循“简单工具，多样组合”的理念，并且相

当小巧。同时，VI 作为单一 UNIX 规范（Single UNIX Specification）的一部分，使得它存在于各种 UNIX 系统中。与上面的图形编辑器相比，使用 Vim 意味着需要多用键盘，少用鼠标。

- **Emacs**: 提供了更强大的扩展功能，它可以称为集成开发环境。与 Vim 相当，Emacs 需要更多的练习才能上手。

注意：如果你是编程新手，不建议使用 Vim 和 Emacs 作为编辑器，因为需要花费大量的时间学习使用编辑器。

2. IDE 和编辑器

在诸如嵌入式这样特定的领域里，由于芯片、开发板等局限，我们只能使用特定的编辑器，或者开发工具。而在 Web 开发领域，可以选择的范围太广了，正是因为有太多的工具可以选择，很容易让我们花费大量的时间在切换工具上。因此，在尝试完编辑器之后，就需要好好练习选定的编辑器。毕竟：

“好的装备确实能带来一些帮助，但事实是，你的演奏水平是由你自己的手指决定的。”

——REWORK

（1）工具是为了效率

寻找工具的目的和寻找捷径是一样的，我们需要更快更有效地完成工作。换句话说，我们可以节省更多的时间去做更多的事情，而这个工具的用途要看具体的事务，如果我们去写一部小说、博客的时候，Word 或者 Web Editor 会比 Tex Studio 来得快。用 TEX 来排版代码、公式会比用 Word 排版的时候来得更快，所以这个工具的好坏是相对的。有时候用一个顺手的工具会好很多，但是不一定会是事半功倍的。我们在使用自带的图形工具就可以完成裁剪、旋转时，就没必要运行 GIMP 或者 Photoshop 完成这个简单的任务。

我们应该专注于内容，在合适的时候使用合适的工具。尽管我更喜欢用 Emacs 作为控制编辑器，但是在服务器上修改某些配置时，我会使用 Vim 来修改内容。而如果是在日常使用过程中，作为日常的暂存区，即暂时放置数据，或者格式化 JSON 文件等适合 GUI 操作时，我会使用 Sublime 作为工具。

(2) 了解、熟悉你的工具

Windows、Word 等软件的功能很强大，只是大部分人只用了其中很少一部分功能。那么，如果我们只用 Word 来写东西，是不可以使用更简单的、开源的 Abiword 来替换它。明显不太可能，因为强大的工具对我们来说有更大的吸引力。

如果你有能力购买你手上的工具，那么就尽可能去了解它能干什么。即使它是一些无关紧要的功能，比如 Emacs 的煮咖啡。如 Linux 下面的命令有一大堆，只是我们常用的只有一小部分——20% 的命令能够完成 80% 的工作。如同 CISC 和 RISC 一样，常用的指令会让我们忘记那些不常用的指令。而那些是最实用的，如同日常工作中使用的 Linux 一样，记忆过多的不实用的东西，不如把它们记在笔记本上实在。

我们只需要了解有哪些功能，如何去使用它，并练习和牢记那些实用的功能。

如我在写本书时，主要是用 Markdown 编辑器在编写内容，最后排版时使用 Word。尽管我持有一个 PhotoShop 的 License，但本书中的许多插图都是由 Word 中的 SmartArt 做出来的，它在创作很多图形时非常便捷，并且容易使用。

经验分享：当我第一次看到有人非常熟练地使用 IntelliJ IDEA 快捷键时，我却花了相当长的时间在练习使用快捷键，如切换、重构等。同样，对于诸如 Vim 和 Emacs 这一类超受欢迎的编辑器来说，我们也需要大量的练习，并且最好有一本相应的书，如在我的书架上有《Vim 实用技巧》和《学习 GNU Emacs》。我也花费了大量的时间在选择编辑器上，选择适合的工具总是有益的。

这里假设读者已经选定了一个开发工具（遗憾的是，这并不是一件容易的事）。现在，我们要着手于提高我们的演奏水平了。

1.2 版本控制

版本控制是记录一个或若干文件内容变化，以便将来查阅特定版本修订情况的一种系统。通常使用版本控制系统来管理代码文件，但实际上，我们会发现除了代码，诸如

文档、服务器配置等也应该进入版本控制系统的管理范围。通过版本控制系统，我们可以做以下事情。

- 将某个文件回溯到之前的状态。
- 将项目回退到过去某个时间点。
- 在修改 Bug 时，可以查看修改历史，查出修改原因。
- 只要版本控制系统还在，就可以任意修改项目中的文件，并且轻松恢复。

常用的版本控制系统有 Git、SVN，但是从近年来看 Git 更受市场欢迎。Git 可以支持分布式、离线使用、本地分支，以及诸如智能的合并能力等特性。

因此，下面以 Git 为例对版本管理系统进行介绍。

1.2.1 Git 初入

如果是第一次使用 Git，则需要设置用户名和邮箱：

```
$ git config --global user.name "用户名"
$ git config --global user.email "电子邮箱"
```

你可以在 GitHub 上新建免费的公开仓库，并按照 GitHub 的文档配置 SSH Key，然后将代码仓库克隆到本地，其实就是将代码复制到你的机器里，并交由 Git 来管理：

```
$ git clone git@github.com:username/repository.git
```

或使用 HTTPS 地址进行克隆：

```
$ git clone https://username:password@github.com/username/repository.git
```

你可以修改复制到本地的代码了（symfony-docs-chs 项目里都是 rst 格式的文档）。当你觉得完成了一定的工作量，想做一个阶段性的提交，并向这个本地的代码仓库添加当前

目录的所有改动:

```
$ git add .
```

或者只是按需要来添加修改的内容:

```
$ git add -p
```

可以输入:

```
$ git status
```

来看现在的状态, 图 1-7 是添加之前的, 图 1-8 是添加之后的情况。

```
fdhuang @test ~/learing/github-roam$ git st
On branch gh-pages
Your branch is ahead of 'origin/gh-pages' by 2 commits.
(use "git push" to publish your local commits)
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   chapters/01-introduction.md
        modified:   chapters/02-github-fundamentals.md
        modified:   github-roam.md
        modified:   index.html

no changes added to commit (use "git add" and/or "git commit -a")
fdhuang @test ~/learing/github-roam$
```

图 1-7 添加前

```
fdhuang @test gh-pages t2 ~/learing/github-roam$ git st
On branch gh-pages
Your branch is ahead of 'origin/gh-pages' by 2 commits.
(use "git push" to publish your local commits)
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   chapters/01-introduction.md
        copied:     chapters/01-introduction.md -> chapters/02-github-fundamentals.md
        modified:   github-roam.md
        modified:   index.html

fdhuang @test gh-pages t2 ~/learing/github-roam$
```

图 1-8 添加后

可以看到状态的变化是从黄色到绿色, 即从 `unstaged` 到 `add`。

在完成添加之后，我们就可以写入相应的提交信息，如这次修改添加了什么内容、这次修改修复了什么问题等。在我们的工作流程里，使用 Jira 这样的工具来管理项目时，也会在 Commit Message 里写上作者的名字，如下：

```
$ git commit -m "[GROWTH-001] Phodal: add first commit & example"
```

这里的 GROWTH-001 就相当于任务号，Phodal 则对应于用户名，后面的提交信息也会写明这个任务的作用。

由于有测试存在，在完成提交之后，就需要运行相应的测试来保证没有破坏原来的功能。因此，可以 PUSH 代码到服务器端：

```
$ git push
```

这样其他人就可以看到我们修改的代码。

1.2.2 Git 工作流

虽然基于 Git 的工作流可能并不是一个非常好的实践，但是这里以这个工作流为参考来进行我们的项目。图 1-9 是由 Vincent Driessen 定义的一个分支策略。

对使用 Git 的新手来说，Git flow 工作流可能有些复杂。下面简单介绍一下。

- 我们平常会工作在开发（即图 1-9 中的“develop”）分支上（通常会直接工作在 master 上，从使用上并没有多大的不同），不同的开发人员可以直接向这个分支提交代码。又或者是当我们在做一些重要的功能时，可能就会从分支上拉出一个新的 feature branches（即，功能分支），等完成后再合并到开发分支上。
- 每个迭代会发布一个新的版本，即使用 release branches 来创建新的标签，这个新的版本将会直接上线到产品环境。那么上线到产品环境的这个版本就需要打一个版本号——这样不仅方便跟踪我们的系统，而且当出错的时候也可以直接回滚到上一个版本。

- 如果在线上有些 Bug 不得不修复，并且由于上线的新功能很重要，就需要一些 Hotfix。

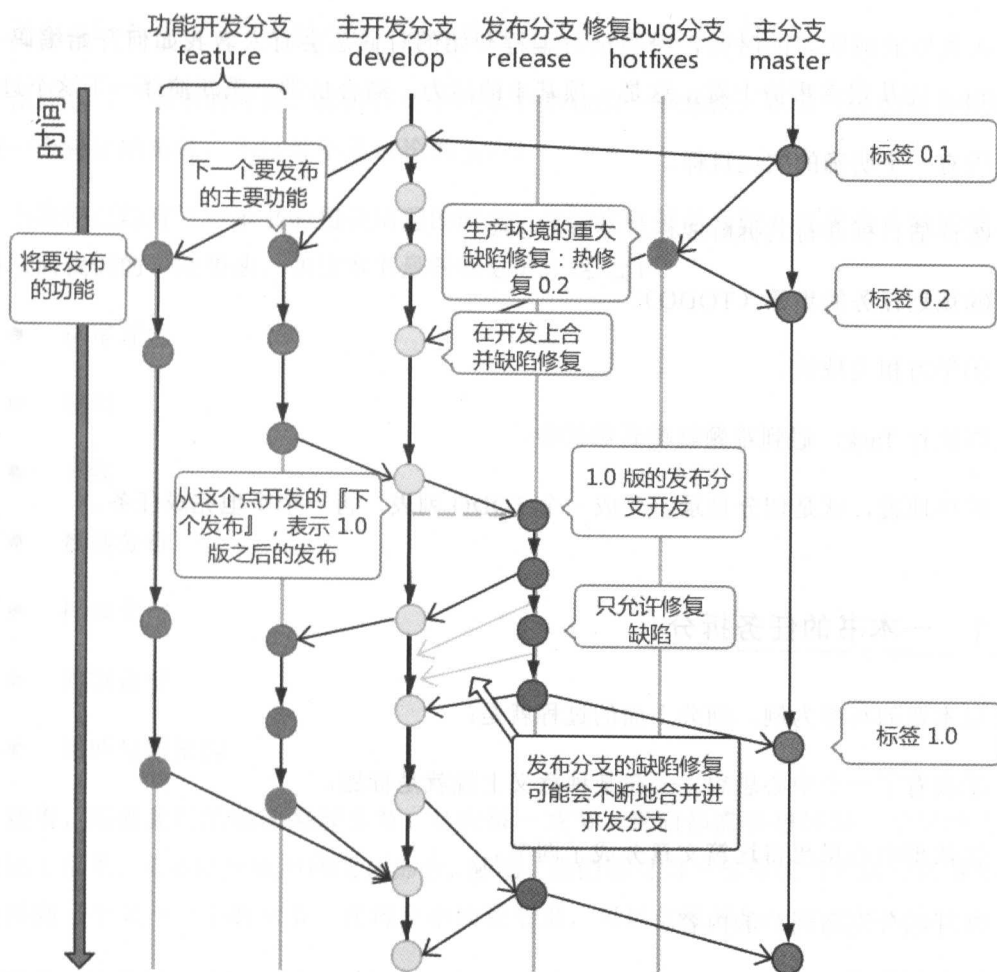


图 1-9 Git 分支策略

从整个过程来看，版本控制起了一个非常大的作用。在开源社区 GitHub 上使用 Git 时，人们会采用 Pull Request 来向一些开源软件提交代码。

1.3 任务拆分

在我毕业前实习的时候，每当结对编程开始的时候总会有人教我如何开始编码——Tasking。而从很多事情上看，这是一项基本的能力。结合日常，重新演绎一下这个过程：

- ①有一个明确的实现目标。
- ②评估目标并将其拆解成任务（TODO）。
- ③规划任务的步骤（TODO）。
- ④学习相关技能。
- ⑤执行 Task，遇到难题就跳到第②步。

简单地说，就是切分目录，变成一个 TODO 列表，再一个个地完成任务。

1.3.1 一本书的任务拆分

以本章的写作为例，细分上面的过程就是：

- ①我有了一个中心思想——在某种意义上说就是标题。
- ②依据中心思想将这篇文章分成了四节。
- ③开始分别写四节的内容。
- ④直到完成。

如果将其划分到一个编程任务，那么也是一样的：

- ①我们想到做一个 xxx 的 idea。
- ②为了这个 idea 需要分成几步，或者几层设计。
- ③对于每一步，我们应该做点什么。

④我们需要学习什么技能。

⑤集成每一步的代码，就有了我们的系统。

现在以这本书的写作过程为例，来看看这个过程是怎么发生的。

在计划写一本书的时候，我们有关于这本书主题的一些想法。正是一些想法慢慢地凝聚成一个稳定的想法，不过这不是讨论的重点。

当我们已经有了一本书的相关话题的时候，我们会打算怎么做？先来个头脑风暴，在上面写满我们的一些想法，如这本书最开始划分了这七步：

- 从零开始
- 编码
- 上线
- 数据分析
- 持续交付
- 遗留系统
- 回顾与新架构

接着，依据我们的想法整理章节。对应每一章节，我们都需要想好每一章里的内容。如在第1章里，又可以分成不同的几部分。随后，我们再对每一部分的内容进行任务划分，就会得到一个又一个小的章节。在每个小的章节里，可以大概策划一下我们要写的内容。

随后，就可以开始写这样一本书——由一节节汇聚成一章，由一章章汇聚成一本。

1.3.2 一个功能的任务拆分

现在，让我们简单计划如何开发一个博客。作为一个程序员，如果要去开发一个博客系统，我们会怎么做？

①先规划一下所需要的功能——如后台、评论、社交分享等，并且还应该设计博客的草图。

②随后就可以简单设计一下系统的架构，如传统的前后端结合。

③进行技术选型——使用哪个后端框和前端框架。

④创建我们的“hello,world”，然后开始进行一个功能的编码工作。

⑤编码时，需要不断地查看、添加测试操作。

⑥完成一个个功能后，就会得到一个子模块。

⑦依据一个个子模块，可以得到我们的博客系统。

与我们日常开发一致的是：需要去划分任务的优先级。换句话说，就是需要先实现我们的核心功能。

对我们的博客系统来说，最主要的功能就是发博客、展示博客。简单地说，一篇博客应该有以下部分。

- 标题
- 内容
- 作者
- 时间
- Slug

然后，我们就需要创建相应的 Model（模型），根据这个 Model 创建相应的控制器代码。再配置一下路由，添加页面。对有些系统来说，就可以完成博客系统的展示了。

1.4 小结

在本章，我们介绍了 Web 开发环境的基本要素，并介绍了如何在不同的操作系统中配置这样的环境。在大部分计算机书籍里，关于搭建环境的内容都不会放在正式章节里，故读者容易忽略这些内容。而当我们真正在编码的时候，却需要去寻找这些工具。

第 2 章

最小可行化应用

在这一章中，我们会以一个简单的 Demo 介绍 Web 应用的模型。然后这个模型将与真实的世界进行类比，并介绍如何简化真实世界的模型，并对其进行建模。依据模型上的一些区别，我们还将引入前 / 后端的概念。

xxx: 我们的 App 上线了, 遗憾的是发现缺少一个重要的页面, 即展示 Growth Studio 相关作品的着陆页。现在只剩下不到一天的时间可以完成这个任务, 你觉得我们应该怎么做?

Phodal: 一天的时间够吗? 一个 Web 应用需要一个数据库和一个 Web 框架等之类的东西吧?

xxx: 不, 这些我们不需要, 我们需要一个能在一天内上线的产品。因此, 我们可以做一个最简单的 HTML 页面。

Phodal: 就是 HTML + JS + CSS, 然后就可以上线了?

xxx: 是的, 我们只需要写写 HTML、JS 和 CSS 就可以上线。然后, 我们将会使用 Bootstrap 这个成熟的前端框架。

Phodal: 为什么不需要稍微具有挑战性的技术?

xxx: 从短期时间来看, 我们正在做的功能虽然很简单, 但是它的业务价值非常高。并且如果我们使用越通用的技术, 它就越容易维护。越快的实现, 就能体现它的价值。尽管对我们的 App 来说, 需要的是一个门户网站, 在首页有一个完整的简介, 带有简单的博客功能, 还要有一个关于我们的介绍。但是现在对我们来说, 最重要的是让用户可以看到有一个首页的介绍页面。

2.1 最小可行化产品

为了尽可能快地做出一个着陆页, 我们能考虑到的最佳方案就是: 基于模板来修改。而在那之前, 我们还需要有一个简单的设计稿。在理想情况下, 我们还应该有一个 Web 设计师来帮助我们设计出原型图。

因此, 让我们重新 Tasking 一下, 如何去做一个着陆页。

- 创建一个原型图。

- 选择一个 Web 框架，优先考虑响应式设计，即对移动设备友好。
- 搭建该框架的“hello, world”。
- 添加需要的功能和代码。
- 打包上线。

随后，再依据需要的功能进行迭代式开发。

下面我们就从绘制一个原型图开始。

通常来说，原型图是由 Web 设计师（或者 UX、UI 等角色）与项目拥有者一起设计出来的。项目拥有者告诉设计师他需要的内容以及大致的样式。设计师在设计过程中需要向拥有者提供一些专业的建议来改进设计，并尽可能让其风格与现有的产品保持一致，以保证基本的用户识别度。

在着陆页中需要下面一些基本元素：

- 页眉：即 Header，包含 Logo 和多个导航按钮。
- 主区域：包含图片轮播，介绍不同项目的一些信息。
- 页脚：即 Footer，主要放置版权信息，当导航的数量较多时，也会放置在这里。

当前我们并没有太多的功能需要实现，只需要创建一个简单的着陆页即可。于是，我们需要先制作一个简单的草图。这样的工具有很多，如 Balsamiq Mockups、Axure，也可以使用 PhotoShop 来实现。

这里推荐一个名为 Pencil 的原型图工具，它是一个开源且免费的原型图工具，可以运行在 Mac OS、Windows 和 GNU/Linux 等不同的操作系统之上。它内建图形管理器、支持使用拖曳，并且可以导出不同的格式；支持使用互联网搜索图形等功能。图 2-1 是 Pencil 的界面，其下载地址为：<http://pencil.evolus.vn/>。

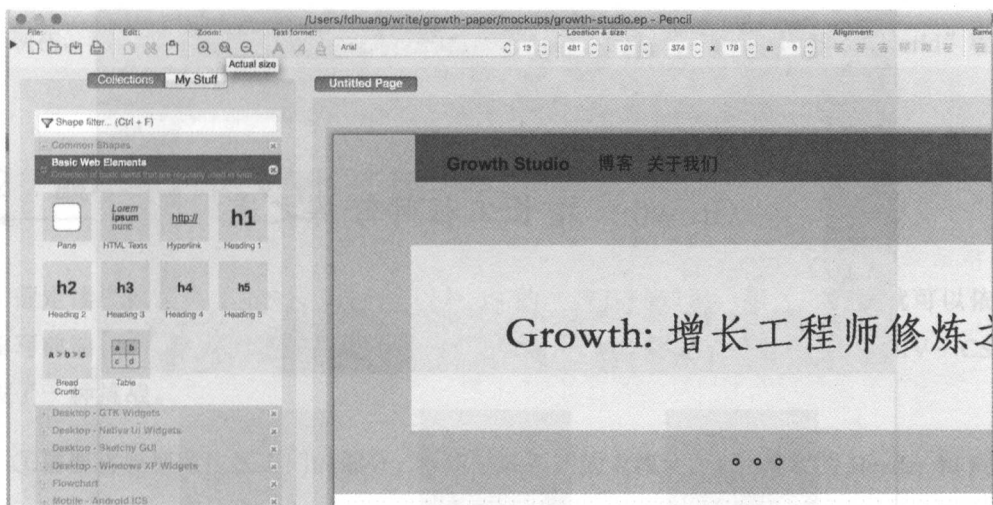


图 2-1 Pencil 界面截图

在 Pencil 的左侧提供了基本的 Web 元素，右侧则是用于编辑的区域。如图 2-2 是笔者花了不到 10 分钟制作出来的一个界面。

多数情况下，我会直接在白纸上绘制，只是不容易修改。要做好设计并不是一件容易的事，如《写给大家看的设计书》所说，我们只需要做到下面四点原则，便可以做一个简单的设计。

- 对比：突出你想让用户看的内容。在图 2-2 里，我们将图片轮播放在最大的地方，用户一进入这个页面便可以找到它。
- 重复：重复出现视觉要素，保持一致性。这点可以体现在字体、颜色、大小等部分。
- 对齐：让元素与元素间有视觉联系。如图 2-2 中几个不同的部分，在左侧和右侧是对齐的。
- 亲密性：让相关的项目归组在一起。如图 2-2 中间的两个小区域，又或者是下半部分的两片介绍区域。

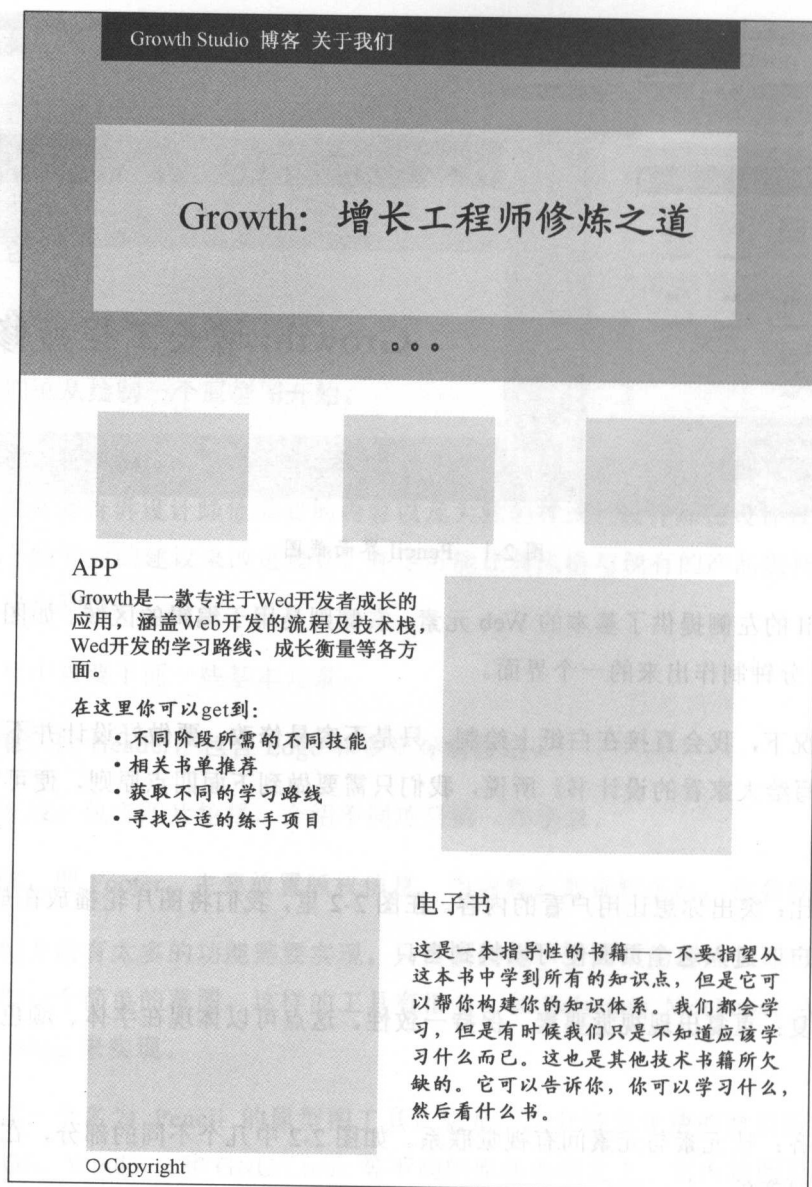


图 2-2 Growth Studio 着陆页

除此之外，你可能还需要一些专业的技能才能做好这方面的设计。这里仅做一个简单的介绍，读者可以自行去学习。

现在，我们就可以进入第二步：选择一个 Web 框架，然后创建一个“hello, world”程序。

2.2 最小可行化 Web 应用

假定读者已经有了 CSS、HTML 以及 JS 的一些基础知识。那么，我们就可以依此图来编写前端代码。当我们是前端新手的时候，要将上面的原型图直接转换为 DIV + CSS 就变成了一种挑战。

①需要将原型图依据不同的部分，将其分成不同的大模块。如，基本的 Header 和 Footer。

②在不同的大模块里又需要将其切分成几个不同的小模块。如，主内容中的图片轮播和项目介绍就是不同的小模块。

③实现细分的模块。

在不同的模块里，我们需要注意分层的概念，即在最底下的内容应该是在 DIV 的最外层，顶层的内容应该是在 DIV 的嵌套层。如，我们应该先给页面的 body 标签设置一个白色的背景，再给导航及图片轮播设置一个灰色的背景，最后给 Header 设置一个黑色的背景。这里的 body 相当于在 DIV 的最外层，而 Header 则是在 DIV 的嵌套里。

尽管这种分层并不会体现在页面里——用户看到的只是效果，但是分层及模块化的好坏影响着代码的可维护性。在采用虚拟 DOM 技术的框架，如 React 里，对于模块化的需求就更高。我们需要尽可能地按照需求对模块进行分解。

2.2.1 使用 Bootstrap 模板

开始之前，我们先简单了解一下 Bootstrap。如果写过一些前端代码，你就应该听说过这个框架。当一个后台开发希望一个前端开发人员推荐一个好用的前端框架时，那么一定有超过 80% 的人说 Bootstrap——来自于 Twitter，在开源社区 GitHub 上拥有 Star 数最多的开源前端框架。Bootstrap 包含了丰富的 Web 组件，如下拉菜单导航条、路径导航、分

页、排版、警告对话框、进度条等。它从 2.0 版本开始支持响应式网页设计，即页面布局可以依据显示网页的设备或者屏幕大小来进行动态调整，并从 3.0 版本开始，将移动设备优先作为设计方针。

由于其在响应式设计方面相当出色，并且在市场上相当流行，我们就将其作为第一个 Web 框架来使用。

Bootstrap 的当前版本是 v3.3.7，读者可以从 <http://getbootstrap.com/> 下载到这个框架。建议读者选择带有源码及示例的 Source Code 下载，即图 2-3 中间部分的 **Download Source**。

Download

Bootstrap (currently v3.3.7) has a few easy ways to quickly get started, each one appealing to a different skill level and use case. Read through to see what suits your particular needs.

Bootstrap

Compiled and minified CSS, JavaScript, and fonts. No docs or original source files are included.

Download Bootstrap

Source code

Source Less, JavaScript, and font files, along with our docs. **Requires a Less compiler and some setup.**

Download source

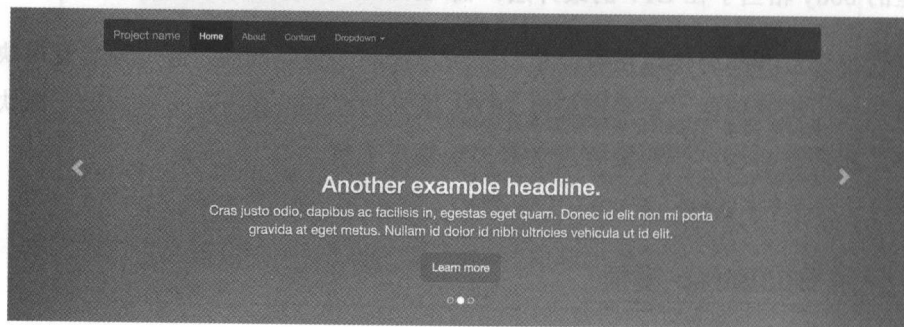
Sass

Bootstrap ported from Less to Sass for easy inclusion in Rails, Compass, or Sass-only projects.

Download Sass

图 2-3 Bootstrap 下载页

解压下载的文件后，可以打开 docs/examples/carousel/index.html，这个文件就是我们想要的模板文件，其效果如图 2-4 所示。



Heading



Heading



Heading

图 2-4 Bootstrap Carousel 模板截图

剩下要做的事情就是从 Bootstrap 的模板里分离出独立可运行的应用。

1. 初始化项目

现在，让我们为这个项目创建第一个 commit。先创建一个目录，如 homepage，随后在该目录下执行 git 的初始化命令：

```
$ git init
```

将初始化这个项目：

```
> Initialized empty Git repository in /Users/fdhuang/write/homepage/.git/
```

接着，就是创建一个 README.md 文件，然后添加该文件并提交，命令如下：

```
touch README.md
git add README.md
git commit -m "init commit"
```

现在我们就可以初始化提交的代码了。

建议读者可以将代码提交到 GitHub 上，这不仅可以在不同的电脑提交代码，还方便我们使用 GitHub Pages 来直接访问我们的网站。

下一步，让我们从 Bootstrap 的示例代码里复制需要的文件。

2. 分离模板

注意：也可以直接下载源码文章，其中的 chapter2 文件夹便是本章的内容。

现在，将需要的文件复制到 homepage 不同的目录下——用于区分不同的文件类型，如 js 目录下放置 js 文件。步骤如下：

①复制 docs/examples/carousel/index.html 到 homepage/index.html。

②复制 dist 目录下所有的目录夹到 homepage/目录下。

③下载 jQuery 到 homepage/js/vendor 目录下。

④复制 docs/examples/carousel/carousel.css 到 homepage/css 目录下。

我们还需要修改 index.html，删除其中不需要的代码。

由于当前并不考虑在不同浏览器下的兼容性，因此，可以删除代码中对于兼容性的处理文件。将 head 中的如下代码。

```
<!-- Bootstrap core CSS -->
<link href="../../dist/css/bootstrap.min.css" rel="stylesheet">

<!-- IE10 viewport hack for Surface/desktop Windows 8 bug -->
<link href="../../assets/css/ie10-viewport-bug-workaround.css" rel=
"stylesheet">

<!-- Just for debugging purposes. Don't actually copy these 2 lines! -->
<!--[if lt IE 9]><script src="../../assets/js/ie8-responsive-file-warning.
js"></script><![endif]-->
<script src="../../assets/js/ie-emulation-modes-warning.js"></script>

<!-- HTML5 shim and Respond.js for IE8 support of HTML5 elements and media
queries -->
<!--[if lt IE 9]>
  <script
src="https://oss.maxcdn.com/html5shiv/3.7.3/html5shiv.min.js"></script>
  <script
src="https://oss.maxcdn.com/respond/1.4.2/respond.min.js"></script>
<![endif]-->

<!-- Custom styles for this template -->
<link href="carousel.css" rel="stylesheet">
```

修改为：

```
<link href="css/bootstrap.min.css" rel="stylesheet">
<link href="css/carousel.css" rel="stylesheet">
```

并将 index.html 文件最后面的代码:

```
<!-- Bootstrap core JavaScript
===== -->
<!-- Placed at the end of the document so the pages load faster -->
<script src="https://ajax.googleapis.com/ajax/libs/jquery/1.12.4/jquery.
min.js"></script>
<script>window.jQuery || document.write('<script src="../../assets/js/ve
ndor/jquery.min.js"></script>')</script>
<script src="../../dist/js/bootstrap.min.js"></script>
<!-- Just to make our placeholder images work. Don't actually copy the n
ext line! -->
<script src="../../assets/js/vendor/holder.min.js"></script>
<!-- IE10 viewport hack for Surface/desktop Windows 8 bug -->
<script src="../../assets/js/ie10-viewport-bug-workaround.js"></script>
```

替换为:

```
<script src="js/vendor/jquery-3.1.1.min.js"></script>
<script src="js/vendor/bootstrap.min.js"></script>
<script src="js/vendor/holder.min.js"></script>
```

如上所示的官方代码里,将页面分成了三部分,这三部分都可以从其 class 名看出来。

①导航条放置在 navbar-wrapper 中。

②图片轮播则放在 carousel 中。

③主要内容则是放在 container 中。在这个 Demo 里,footer 是放在 container 中,而不是一个独立的 Div。

我们已经粗略地完成了“hello,world”程序,现在只需要对其进行完善即可。进入下

一步之前，先对如下代码进行提交：

```
git add .  
git commit -m "init project"
```

2.2.2 完善原型

由于这是一个很匹配我们需求的模板，我们只需要修改代码中相应的内容即可。

1. 修改页面标题

将代码中 title 标签的内容修改为<title>Growth Studio - Enjoy Create & Share</title>。

2. 修改导航

将 Project name 修改为 Growth Studio，并修改代码中不同导航栏的名称。

3. 修改轮播图片及介绍部分

只需要添加我们所需要的文本到相应的文字块即可。

4. 修改 Footer 部分

将如下代码：

```
<footer>  
<p class="pull-right"><a href="#">Back to top</a></p>  
<p>&copy; 2016 Company, Inc. &middot; <a href="#">Privacy</a> &middot;  
<a href="#">Terms</a></p>  
</footer>
```

修改为

```
<footer>  
<p class="pull-right"><a href="#">回到顶部</a></p>  
<p>&copy; Growth Studio</p>
```

```
</footer>
```

Done!让我们再将上面的代码添加到版本控制里:

```
git add .  
git commit -m "update content"
```

如果我们有一个对应的故事卡, 就可以这样来写提交信息: [card-xx] update content。

如果这是一个多人合作的项目, 就应该将我们的名字加上, 如: [card-xx] Phodal: update content。

由于着陆页只有一个页面, 并且函数功能都依赖于框架, 所以不需要对这些代码进行测试。

2.2.3 简单上线

制作完着陆页后, 接下来需要上线。如果你想尝试在服务器上运行你的网站, 则需要:

- 找到一个域名服务商, 并注册、购买一个域名。
- 租用一个云服务或者虚拟主机。有一些云服务提供商 (如 Amazon AWS 云服务) 会提供一年的免费试用。

注册、购买域名是必需的, 至于云服务, 我们则更推荐先使用免费的。因篇幅原因, 这部分内容略去不谈。因此, 我们先介绍如何使用 GitHub Pages 来发布我们的网站。

1. 使用 GitHub Pages

事实上, 如果你熟悉 Git 和 GitHub, 只需要在命令行里执行如下语句:

```
git checkout -b gh-pages
```

来创建一个 gh-pages 分支, 然后向 GitHub 提交这个分支即可:

```
git push origin gh-pages
```

访问在 GitHub 后台的设置 (Settings)，就可以在 GitHub Pages 上看到类似于图 2-5 所示的内容。

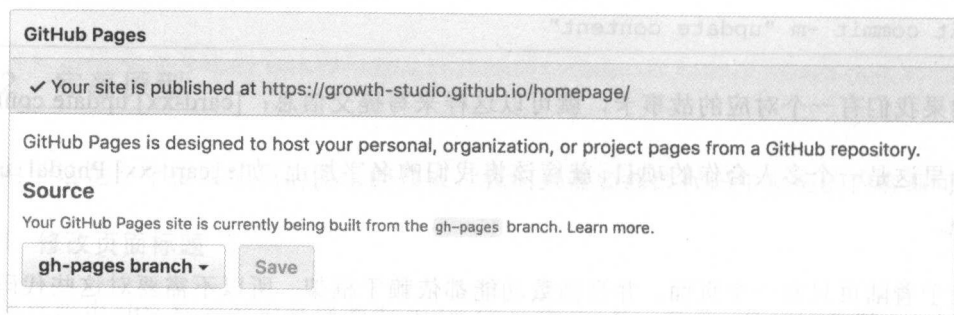


图 2-5 GitHub Pages 显示 URL

这里的 URL 是根据下面的规则生成的：用户名或者组织名.github.io/项目名，如 `https://growth-studio.github.io/homepage/`。

如果读者有一个域名，只需要在域名解析服务器上添加相关的 CNAME¹ 域名解析，将其指向你的用户名或者组织名.github.io，并在项目文件里添加对应的 CNAME 文件，在 CNAME 里添加我们想要指定的域名。

图 2-6 是笔者拥有的域名 **growth.ren**，只需要添加在主机记录 **studio**（即要使用的二级域名）中。接着，在记录值里填上 **growth-studio.github.io** 即可。

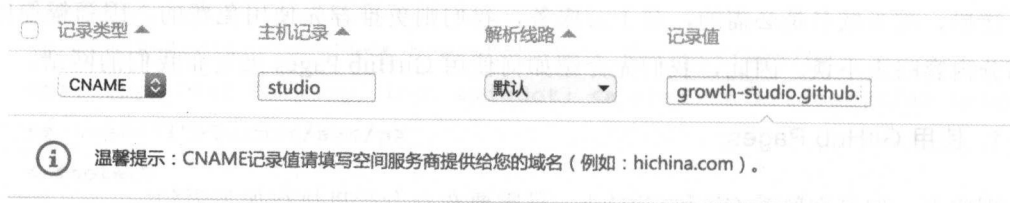


图 2-6 Growth Studio CNAME 添加

最后，在我们的项目里添加一个名为 CNAME 的文件，在文件中写入 **studio.growth.ren**。

¹ CNAME 记录，即：别名记录。这种记录允许用户将多个名字映射到同一台计算机。

等几分钟后，就可以直接访问 `studio.growth.ren` 来查看我们的网站了。

现在，我们只需要访问这个 URL 就可以访问网站了，是不是很酷。

2. HTTP 服务器

当我们访问 `studio.growth.ren` 域名的时候，在服务器端将有一个对应的 HTTP 服务器来对请求进行处理。现在，让我们来搭建一个简单的 HTTP 服务器模拟服务器环境，用它来处理来自其他计算机访问的 HTTP 请求。

因为已经安装有 Python 的环境，就可以使用 Python 库中自带的 HTTP 模块 `SimpleHTTPServer`。在 `homepage` 目录下，运行下面的命令：

```
python3 -m http.server
```

就可以运行一个 HTTP 服务器，端口是 8000。

```
Serving HTTP on 0.0.0.0 port 8000 ...
```

因为在目录里有一个 `index.html` 文件，该文件将成为默认主页。因此，我们可以直接打开 `http://localhost:8000` 来访问网站。

读者如果使用的是 Python 2.7 版本，可以直接用下面的命令：

```
python -m SimpleHTTPServer
```

我们也可以使用局域网内的其他手机、平板、电脑等上网设备来访问创建的这个网页。

除此之外，使用嵌入式服务库 `Mongoose` 也是一个不错的选择，这是一个易于使用的跨平台 Web 服务器，可以嵌入到其他应用程序中。使用包管理工具在电脑上安装这个软件，并运行 `mongoose`，就会在终端看到下面的启动过程：

```
Starting web server on port 8000
```

为了更具有真实感，读者可以使用如 `Raspberry Pi` 这一类的微型计算机，或者将其他计算机当成服务器。

这里只使用简单的 HTTP 服务器，我们将在第 6 章引入 Nginx，并使用其作为服务器。

3. 从浏览器到服务器

下面使用一句命令行来探索：当我们访问一个网站时，其背后都发生了什么。我们将使用 cURL 来做这次探索，它是一个利用 URL 语法在命令行下工作的文件传输工具。读者可以使用包管理工具来安装这个软件，或者从 <http://curl.haxx.se/download.html> 处下载到。执行下面的命令（参数 -v 可以显示一次 http 通信的整个过程）：

```
curl -v http://studio.growth.ren
```

就会看到下面的响应过程：

```
* Trying 151.101.100.133...
* Connected to studio.growth.ren (151.101.100.133) port 80 (#0)
> GET / HTTP/1.1
> Host: studio.growth.ren
> User-Agent: curl/7.43.0
> Accept: */*
>
< HTTP/1.1 200 OK
< Server: GitHub.com
< Content-Type: text/html; charset=utf-8
< Last-Modified: Tue, 18 Oct 2016 23:26:01 GMT
< Access-Control-Allow-Origin: *
< Expires: Wed, 19 Oct 2016 09:51:52 GMT
< Cache-Control: max-age=600
< X-GitHub-Request-Id: 2BF94820:1173:22DB5443:58073FE0
< Content-Length: 8331
< Accept-Ranges: bytes
< Date: Wed, 19 Oct 2016 09:41:52 GMT
< Via: 1.1 varnish
< Age: 0
< Connection: keep-alive
```

```

< X-Served-By: cache-nrt6123-NRT
< X-Cache: MISS
< X-Cache-Hits: 0
< X-Timer: S1476870112.700193,VS0,VE183
< Vary: Accept-Encoding
< X-Fastly-Request-ID: 123a4fbbc5121744a82a66e2d98b66b40b0e4922
<
<!DOCTYPE html>
<html lang="en">
<head>
...

```

我们尝试用 cURL 去访问 **studio.growth.ren** 时，会根据访问的域名找出其 IP，通常这个映射关系来源于 ISP 缓存 DNS（Domain Name System）服务器[^DNSServer]。

- 以 “*” 开始的前 8 行是连接相关的一些信息，称为**响应首部**。我们向域名 **studio.growth.ren** 发出了请求，接着 DNS 服务器告诉我们网站服务器的 IP，即 151.101.100.133，连接的端口是 80。
- 以 “>” 开始的内容是我们向 HTTP 服务器发送请求。
- Host 对就是我们要访问的主机域名，GET / 则代表要访问的是根目录，紧随其后的是 HTTP 的版本号（HTTP/1.1）。
- User-Agent 通常指向的是使用者行为的软件，通常会加上硬件平台、系统软件、应用软件和用户个人偏好等信息。Accept 则指的是告知服务器发送何种媒体类型。
- 以 X-** 开始的部分则表明 GitHub 对于网页有缓存处理，它使用的缓存服务器 Varnish 版本是 1.1。
- <!DOCTYPE html> 则是我们接收到的 HTML 内容。

在解析域名 DNS 的时候，需要先从本地 DNS 服务器查询。如果没有，再向根域名

服务器查询——这个域名由哪个服务器来解析，直至最后获取真正的服务器 IP 后，才开始下载页面的内容。

根据 HTML 中的 script 和 link 标签获取 JavaScript、CSS，并获得相应的 HTML、JavaScript、CSS 后，浏览器就开始渲染这个页面。有兴趣的读者可以阅读 WebKit 相关的书籍，了解浏览器是如何一步步渲染出完整的页面的。

2.3 精益与敏捷软件开发

你可能会对此很诧异，为什么要用这么简单的东西来实现这个功能，而不是使用高大上的技术。这是因为我们在尽可能快地推出产品，并收集市场的反馈来改进我们的产品。为了实现这一点，我们需要结合精益和敏捷开发的思想，如《精益思想》一书所说：

精益思想的核心就是（消除浪费）以越来越少的投入——较少的人力、较少的设备、较短的时间和较小的场地创造出尽可能多的价值。同时，也越来越接近用户，提供他们确实要的东西。

精益思想可以让我们一步步做出符合市场需求的产品，并将其做到最好，而敏捷软件开发则可以让我们应对这个过程中的需求变化。

2.3.1 敏捷软件开发

这里我们将展开对敏捷软件开发的介绍，不同的软件开发方式都有其适用范围，敏捷软件开发主要想解决的问题是应对快速变化的需求。现在看来，它更适合于互联网领域的软件开发。与使用迭代式开发的软件开发流程相比，敏捷软件开发具有更短的迭代周期，这样做可以更快速地应对需求变化。敏捷宣言包含以下方针。

- 个体和互动：高于流程和工具。
- 工作的软件：高于详尽的文档。

- 客户合作：高于合同谈判。

- 响应变化：高于遵循计划。

个体和互动所强调的就是在团队内部要经常沟通，沟通越频繁，对系统和设计的理解越有帮助。工作的软件则侧重于我们要做出一个真正可用的软件，这就意味着，我们要尽可能快地推出一个原型，如 `hello, world`，再在此基础上一步步添加所需要的功能代码。客户合作强调深入与客户合作，而不是合同本身，如有时客户并不是真实想要这样的软件。响应变化即市场的需求一直在变化着，如果我们不能快速应用需求变化，就会做出不适合当前市场的功能。

每个人或每个对敏捷软件开发的理解都有所不同。尽管在实践上略有差异，但是这些差异有可能是由于团队的自我改进而促成的。

1. 沟通

在敏捷团队里，人们强调沟通的重要性。当开发人员得到一个新任务时，他就需要和测试人员一起从业务分析人员那里理解清楚业务的需求，并且在实现过程中可以不断地确认需求是否正确。这样做可以确保开发人员做出来的功能是业务所需要的，同时也确保了测试人员和开发人员、业务人员对需求的理解是一致的。相似的，业务分析人员需要与项目经理一起确定好每个迭代需要做的功能，并且与业务人员或者相关人员一起确认好需求。由一步步自上而下的沟通来保证对需求的理解是正确的，从而避免做出不正确的需求与功能。需求的变化反映了一个团队协助的好坏。尽管开发人员对于变化的需求不是很高兴，但是其对最后交付的软件总是有价值的。

沟通与合作也体现了人在软件开发过程中的重要性。对于需求的沟通，要求人们在开发过程中有更好的合作能力。例如，在日常工作中，我们需要：

- 每日站会。我们会讲述“昨天做了什么，今天做了什么，遇到什么问题。”，这时我们主要关注于是否遇到问题，这个问题是否会影响到这个迭代的交付。这也是每个人每天都会和其他人沟通的开始。
- 结对编程。尽管结对编程存在相对比较高的成本，但两个人来实现一个功能，它

除了可以减少 Bug，还能分享项目相关的知识。它特别适合有新人加入团队时，帮助新人熟悉项目、团队编程风格、分享知识。对有经验的人来说，它需要更好的沟通和引导才能促使新人成长。

- 代码检视（或者称代码审查，即 Code Review）。代码检视除了用于改善软件质量之外，还侧重于与别人分享编程知识，推荐一些编程实践。同时，还可以帮助新人提高编程能力与敏捷实践。对新人来说，代码检视是一种痛苦的体验，但是它有利于提高自己。如果在检视过程中遇到不同观点的讨论，也能增加团队的知识。
- 开卡和验收。这些正是我们在上面讨论的对于需求的沟通，开发人员和业务分析人员、测试人员一起开卡来进行新的任务，并在最后一起完成验收，以达到需求的标准。
- 迭代回顾。在敏捷团队里，迭代的最后一天，会对迭代中的实践进行回顾——哪些做得好，哪些做得不好。它可以帮助我们解决迭代过程中遇到的问题，以及如何更好地解决问题。

这些实践都依赖于团队成员，他们需要更好地表现自己内心的想法。当一个团队成员在初期表达遇到问题时，应该努力帮其改善沟通问题。而对沟通问题的改善也可以帮助我们改善软件质量。

2. 软件质量

真正决定一种开发方式好坏的不是方式本身，而是最后交付的软件质量。在对传统软件开发的一些问题进行总结之后，开发者们提出了一系列的观点和实现来解决并改善这些问题。除了在上面提到的代码检视和结对编程，我们在开发过程中还采用了下面一些策略：

- 测试驱动开发（Test-Driven Development，简称 TDD）。即在写代码之前，先写测试，随后再实现功能代码，最后对代码进行重构。在前端领域，在 UI 的测试使得其 TDD 带有副作用。在后端领域，它显得非常快速并且有帮助。它可以驱

使我们写出足够小的类，并且可以保证代码都被测试覆盖。

- 重构。对代码进行重构，在不改变结果的情况下，使其更加简单、易读，则可以减少遗留代码的产生。
- 技术债管理。由于发布截止期限的临近，我们可能在这次发布里使用一些比较粗暴的方式来实现功能，又或者是由于时间及人力的限制，使得我们的代码库里拥有一些不太好的代码。每当出现一点时，我们就对问题进行标识，在适当的时候修复它。如果我们所欠的债务越来越多，就意味着软件的质量受到影响，需要重新审视这个问题。
- 浮现式设计。敏捷软件开发并不意味着架构设计不存在，在实践过程中，我们依据问题（即上面提到的内容），对代码进行改进。随着我们对软件质量的重视，我们的架构就会浮现出来。

对好的工匠而言，可工作的软件是不够的，还要做得更好。

3. 持续改进

敏捷软件开发里，最大的循环周期便是迭代。在迭代开始与结束的时候，我们会进行以下工作。

- 迭代计划会议（Iteration Planning Meeting, IPM）。全体人员一起了解要开发的任务，并对任务进行估算等。随着迭代的进行，我们很容易估计一个迭代可能完成的任务的工作量，并分配好这个迭代所需要的工作量。
- 迭代回顾会议（Retrospective）。在这里，我们对这个迭代进行回顾，找到做得好的、不好的、需要改进的，并保持好的习惯，对不好的内容进行改进。它可以让我们关注于团队和项目存在的问题，使我们持续改进团队及项目交付。

除此之外，在每天的站会、代码检视里，我们也在持续分享和传递知识，使知识传递越来越充分。

这些是基于平时的实践总结而出，并推荐的一些方式。因此，只是一个简单的介绍，

更多详细的内容可参见：《敏捷软件开发》、《学习敏捷》等敏捷软件开发相关书籍。

2.3.2 精益

精益是最近几年非常流行的一个词，人们在各种地方讨论精益，讨论如何充分利用好资源、如何尽快地交付软件等。

1. 精益软件开发

与敏捷软件开发相比，精益软件开发对团队和开发人员有更多的能力需求，同时它也存在以下一些原则。

- 消除浪费。
- 增强学习。
- 尽量延迟决定。
- 尽快发布。
- 下放权力。
- 嵌入质量。
- 全局优化。

敏捷软件开发明确要求团队的分工合作，这就使得当团队的某一个角色不在时（如业务分析人员或者测试人员休假时），团队的交付就可能因此而受阻；或者在迭代快结束时，团队的交付压力可能就转接到测试人员身上。这时就存在生产力上的浪费，为了保证交付的进行，就需要有人可以临时代替该角色。然而这就要求团队有更好的学习能力，人们开始依此来打造全功能的团队。同时，增强对敏捷软件开发中迭代的改进。

由于其对于人员能力及团队、组织的依赖较强——需要自上而下的改变，在此就不展开详细讨论了。

2. 精益环路

以 App 为主导的项目里，开发人员通常会制作出 App 的着陆页，用来介绍 App 相关的资料，以此来激发用户对 App 的使用欲望。而作为着陆页本身并不需要太复杂的功能。我们只需要制作出这样的页面，并尽可能多地收集用户想要的功能和反馈。而尽快发布正是精益软件开发的一部分，同时也是精益思想的体现，如图 2-7 所示。



图 2-7 精益环路

根据我们对市场需求的想法创作出原型，推出产品、接受用户的反馈。依据收集到的数据，对产品进行改进，再推出新的功能。而当用户不需要这个产品或者功能时，我们可以转而实现其他功能。在这个迭代循环里，我们可以做出用户真正想要的产品，并逐步做到最好。在这个过程中，我们可以在不同的地方采用精益的思想：用户体验数据、软件开发、数据分析等。

本书在编写过程中也遵循了相似的环路：先推出免费电子版的书籍，接收用户反馈。依据用户的反馈添加需要的章节，不断完善，最后才有了此书纸质版本的编写，如图 2-8 所示。

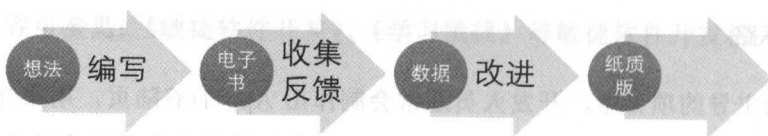


图 2-8 精益出版

同样，我们也将这个思想带入了代码的开发中——推出原型、上线、添加功能、继续上线等，这些内容都会在本书后面部分讨论。除此之外，以下这些内容可以帮助我们改进迭代及交付的软件。

- 数据分析。
- 持续交付。
- 回顾。

2.4 小结

让我们回顾一下我们学到了什么内容？

本章从设计一个原型图开始，展示了如何依据原型图来开发一个网站的着陆页，以及如何简单上线这个产品。同时，我们以精益及敏捷软件开发的思想来说明为什么要这样做。一个简单的网站制作流程如下。

- 先写一个“hello, world”程序。
- 实现完整的上线流程。
- 继续编写功能、上线。
- 持续交付功能。

另外，我们还需要选择一种好的语言、一门好技术和一个好的框架。

第3章

技术选型与业务

在上一章中，我们快速制作了一个静态的着陆页。由于它并不依赖于后台，因此能方便地改造成适合于其他后端语言的页面，也可以将其当成独立的页面部署。而在本章，我们需要选择后台语言、后台框架来完善系统。我们还将依赖于我们的业务来进行技术选型。从业务价值与技术价值的坐标中选择出更符合当前业务需要的技术栈。

xxx: 我们已经上线了首页, 效果还不错。业务人员说还需要有一个可以发布内容的博客系统来发布一些新的动态, 他们希望这个产品能在一星期内上线。

Phodal: 一星期, 这么快? 时间来来得及吗?

xxx: 可能来得及, 只要我们选择好技术方案。

在上一章里, 我们仍然经历了同样的问题。

- ①我们正在做一个很重要的功能, 然而他的技术复杂度并不高, 但是业务价值很高。
- ②由于时间紧迫, 我们手上最合适的方案就是最拿手的那个方案。
- ③基于业务的编码, 需要考虑业务价值与实现难度的关系。

如果是你, 你会怎样选择?

我们的目标是: 一个符合完整的前后端 Web 开发项目。对此进行 Tasking, 应该就会有:

- ①设计一个符合需求的架构。
- ②选择符合需求的前端、后台框架。
- ③使用上一步里的框架来创建一个“hello, world”程序。
- ④合并在上一章开发的着陆页到项目里。
- ⑤完善开发构建、流程部署等。
- ⑥一步步开发需要的业务功能。

我们将在本章详细介绍第①~④步。现在, 我们简单了解一下如何进行架构设计。

在设计一个 Web 应用的软件架构时, 要考虑的基本技术因素有。

- 如何进行技术选型。
- 如何运行和部署应用。

而当我们真正要一个产品时，则需要考虑：

- 系统的安全性，如认证、加密策略。
- 系统的稳定性、可靠性、可用性。
- 异常及错误处理。
- 数据收集，如记录用户的关键操作。
- 上线出问题时的应对策略，如回滚等。

在需要与其他系统集成时，还需要考虑：

- 如何进行模块划分，以及模块间如何通信。
- 如何保证第三方系统异常时还正常工作。
- 如何监测第三方系统的数据。

除了这些，还有估算和如何应对需求变化等一系列细节。上述每一部分都是一个大的话题，这里只讨论基本的技术因素：技术选型。

3.1 技术选型

尽管 Web 开发的技术在过去十几年里发生了翻天覆地的变化，但是其思想和本质仍然是一样的：仍需要先由浏览器客户端向服务器发生请求。只是对于架构的设计已经不是由服务端（或称后端）来驱动——在早期的系统架构设计里，我们在服务器端处理一切请求，在服务器端保存或者临时保存用户的状态。

在过去，从浏览器里提交一个表单，这个表单的校验将交由系统后台来进行，随后再返回结果并显示到浏览器上。而在今天，我们会先使用 JavaScript 对表单进行校验，校验通过后才发给后端来进行处理。

客户端变了，从原来单一的桌面浏览器变成了手机、桌面、平板等其他设备，从原先对 1024×768 分辨率的支持，扩展到了对 1440×900 、 960×480 等不同分辨率的支持。对于客户端界面 UI 的设计，由单一的网格设计变成了响应式设计——可以应对屏幕大小变化来调整分辨率。

在客户端上有更好的用户体验，我们也改变了这其中的设计来应对变化。我们将越来越多的可以由客户端处理的逻辑移到了浏览器上，简化了系统后台对不必要的业务逻辑的处理。前台与后台的交互变成了一个没有状态的 API，API 设计变成了相当重要的一个环节。

在服务器端也发生了一些变化。为了应对前端和移动端对 API 的需求，后端的开发压力开始转向 API 设计，设计符合需要的 API——如超媒体，开发者开始使用微服务的架构来代替现有的单体应用，以换取更好的快速开发能力。

我们在绪论里提到了技术选型的一些因素，如图 3-1 所示。

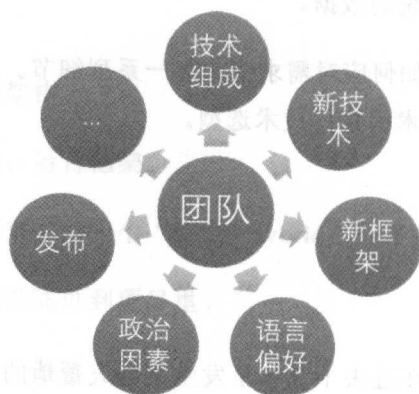


图 3-1 技术选型考虑因素

在这些因素里，需要侧重考虑团队因素。团队的成员是真正的实施人员，也决定了最后软件交付质量的好坏。因此，如果在一些外部因素可控制的情况下，应该优先考虑团队内部的声音，如使用新技术与否。使用新技术就会存在技术风险，而作为一个开发者，总是希望自己使用的技术可以跟上潮流，它可以让团队向学习型团队转型，并且持续使用旧的技术，容易影响整个团队的气氛，这也是信心不够的一种体现。

除此之外，不同的人对框架所能提供的功能也有不同的需求。一般来说，在一种语言里，如果有一个框架 A 自身包含大部分的功能和扩展。那么就会有一个框架 B 提供核心功能，并以扩展的形式将一个个组件进行拆分。

在决策的过程中很容易导致分歧，特别是当可选方案越多时，最后存在的争执会越大。不过，在这里不会存在这样的问题，下面来看看我们可以选择的一些技术方案。

遵循这些基本点，以及本章开头时所需要的功能，我们所需要的就是：

- 一个后台 MVC 框架，对 CMS 支持友好。
- 支持响应式及移动设备的前端 UI 框架。

3.1.1 后端选型

在进行后端选型的时候，要选择的实际上是一个框架。后端领域所使用的技术和框架已经趋于稳定，我们只需要框架。当有多个框架适合时，再选择适合的语言。不得不指出的是，当我们喜欢一种语言的时候，我们可能会偏爱于在这门语言里寻找可用的方案。这自然是有好有坏，好的一点是：我们可以成为这门语言的专家；不好的一点是：选择的可能不是最合适的方案。

下面先探索一下可用的语言，以及它们对应的 Web 框架（以下内容均为个人观点）。

1. JavaScript

按照当前的流行趋势来看，JavaScript 是一门性价比非常高的语言。因为只要是 Web，就会有前端，只要有前端，就需要有 JavaScript。与此同时，Node.js 在后台中的地位已经愈发重要了。对一般的项目而言，可以使用它来完成前端和后台，除此之外，还有移动应用。

在那些可以使用浏览器来运行的设备上，我们都可以使用 JavaScript 来开发使用，例如：

- 使用 Node.js 作为后台语言，Express、Koa 等作为后台 MVC 框架，再选择一个前端框架来实现前台。
- 使用基于浏览器内核的桌面应用 Electron，加上 Node.js 生态系统里的模块来实现桌面应用。
- 使用混合应用移动框架 Cordova，混合应用框架 Ionic 来实现跨平台的移动应用。
- 使用 Tessel 和 Ruff 等硬件来开发移动应用。

人们使用 WebView 和 JavaScript 来开发应用的很大一部分原因是成本比较低。除了可以高效地开发 UI，还支持跨平台运行，即只需要编写一次代码就可以在不同的操作系统上运行，并且当应用对性能要求不高时，只要适当地优化，它就可以表现得相当不错。

在这门语言里，有两个后台 MVC 框架比较流行。

- Express: 是在 Node.js 上最早的 MVC 框架，它由 Ruby 上的轻量级框架 Sinatra 启发而来的。其框架本身封装了大量实用的功能，核心特性是使用中间件来处理 HTTP 请求。
- Koa: 是由 Express 的核心开发者基于 ES6 新特性打造的新框架。与 Express 相比，去除了一些框架自带的功能，更加轻量级，可以让开发者有更多的选择。当然，这也意味着需要用户自己去搭建这些环境。

简单对比一下两者，Express 发展得比较早，其生态系统比较丰富，很容易找到所需要的插件。Koa 则基于 ES6 语言带来一些新的特性，实时解决旧语言的一些问题，如回调等。

2. Python

Python 诞生得比较早，其语言特性是做事情只有一种方法，这个特点也决定了这门语言很简单。与 JavaScript 相比，它仍是一门性价比非常高的语言，只是它不能在前端运行。

Python 是一门简洁的语言，有大量的数学、科学工具、人工智能的库，这意味着在不远的将来它会发挥更大的作用。同时在 Web 开发领域也有广泛的应用，除了正常的 Web 开发，它还在网络爬虫中广受欢迎。

同样，在 Python 语言里也有两个不错的框架可以选择，其中的 Django 是重量级框架，Flask 则是轻量级框架。

- Django: 最早是被应用于内容管理系统而开发的，其框架里自带了相当多的组件：ORM、表单序列化及验证系统、后台系统、缓存框架、中间件支持等。在其官网上宣称是：The Web framework for perfectionists with deadlines，它既可以满足完美主义者，又可以在截止日期前交付软件。
- Flask: 是一个轻量级的框架，它只有简单的核心部分。换句话说，你可以按自己的需要添加 ORM、用户认证、文件上传等功能。在今天来看，它的生态系统也相当丰富，可以完成绝大部分功能。

选择 Flask 而不是选择 Django 的原因是：Django 本身规定好了一系列的规范和习惯。因而在编程时，我们只需要按步骤一步步往下走即可。

本书采用 Django 作为 Web 开发框架的主要原因是，它适合作为 CMS 框架，并且提供了丰富的组件功能，如用户权限管理、自带后台管理系统、ORM 等。

3. Java

在今天看来，Java 仍然受企业欢迎，除了在企业级 Web 系统开发上，它还在 Android 应用的开发上绽放光彩。

在校期间，笔者一点儿也不喜欢 Java。后来才发现，我从 Java 中学到的东西比从其他语言中学的东西还多。如果 Oracle 不毁坏 Java，那么它会继续存活很久。我可以用 JavaScript 造出各种我想要的东西，但是通常我无法保证它们是优雅地实现。过去人们在 Java 上花费了很多时间，或在架构上，或在语言上，或在模式上。由于这些投入，都给了人们很多启发。这些都可以用于新的语言和新的设计，毕竟没有什么技术是独立于旧的技术产生的。

由于在 Java 语言里，笔者主要接触的是 Spring 框架，因此下面讨论一下 Spring。

- Spring MVC: 是由 Spring 框架提供的构建 Web 应用程序的全功能 MVC 模块。由于框架本身高度可配置，即可以直接使用编写 XML 而不是 Java 来实现

功能。它是一个典型的 MVC 框架，并且也是一个纯正的 servlet 系统。

- **Spring Boot:** 其作用在于创建和启动新的基于 Spring 框架的项目。系统本身做好了针对不同框架的配置与集成，我们只需要对其配置，并编写少量的代码即可。

如果你正在考虑使用 Spring 框架，建议使用 Spring Boot。

4. PHP

PHP 是一门很容易上手语言，由于其容易上手，并且发展得比较成熟。因此，有相当多的个人网站使用它作为开发语言，如 Facebook 这样大流量的网站也在使用它。另外，不得不提及的是 WordPress 已经占领了 CMS 市场超过一半的份额，并且它也占领了全球网站的四分之一。WordPress 原生是为博客系统而创建的开源框架，由于博客系统和内容管理系统在功能上很多是相似的，因此它成了最具知名度的内容管理系统（content management system CMS）。

在这里并不基于 WordPress 来开发内容管理系统，因为它已经是一个相当成熟的框架了。如果你需要一个博客系统或者内容管理系统，首选 WordPress，然后才是自己编写。

PHP 在框架方面有比较多的选择，遗憾的是，笔者只对 Laravel 比较了解。它是在 PHP 5.3 之后开发的新框架，其类似于 Ruby on Rails——为 PHP 程序员提供快速开发的机制——提供快速开发的工具集，如生成代码、数据迁移、ORM 等。

5. Ruby

Ruby 是一门优美而巧妙的语言，它可以使编写出来的代码看上去更自然、简洁，更具有表达力，因此深受程序员欢迎。早期 Ruby 语言的应用场景特别少，直至 Ruby On Rails 的出现。它是严格按照 MVC 结构开发的 Web 开源框架，其致力于提升程序员的快乐感和生产效率——快速创建页面、模板和查询功能等。不过如今由于可维性和性能的问题，它正在逐渐被替换。只是对初创公司来说，它的开发效率仍使得它是一个不错的选择，随后在业务稳定后使用其他框架来替换。

同样，由于 Ruby On Rails 是一个重量级的选择，Ruby 程序员也推出了自己的轻量级框架 Sinatra。它是一个基于 Ruby 语言的 DSL（领域专属语言），由于其代码行数少，且

简单、简洁，可以很容易深入理解框架并对其做出定制。

6. 其他

除了上面提到的语言，还有一些语言也很不错，如 Go、Scala 等。限于作者能力有限，以及篇幅的原因，在此就不展开详细讨论。

选择框架和语言后，还需要在选型的时候考虑数据存储的问题。

3.1.2 数据持久化

信息源于数据，我们在网站上看到的内容都应该是属于信息的范畴。这些信息是应用从数据库中根据业务需求查找、过滤出来的数据。数据通常以文件的形式存储，毕竟文件是存储信息的基本单位。而数据持久化除了将数据存储到存储介质上，还会将数据及其关系模型存储到其中。只是由于业务本身对 Create、Update、Query、Index 等有不同的组合需求，就引发了不同的数据持久化方案。

对后台开发人员来说，对数据进行持久化有很多种选择。

①文件存储。

②数据库。

③搜索引擎。

这些数据都可以被创建、读取、更新、删除，即 CRUD。

1. 文件存储

通常来说，以这种方式存储最常见的方式是 log（日志），如 Nginx 的 access.log。像这样的文件就需要一些专业软件，如 GoAccess 或者 Hadoop、Spark 来做对应的事。

在数据库出现之前，人们都是使用文件来存储数据的。数据以文件为单位存储在硬盘上，并且这些文件不容易一起管理、修改等。如图 3-2 所示的是笔者早期存储文件的一种

方式。

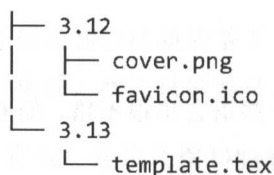


图 3-2

每天我们都会修改、查看大量不同类型的文件。而由于工作繁忙，我们可能没有办法对这些文件进行分类。有时选择的是先按日期划分文件，接着在随后的日子里归档。而这种存储方式依赖人来索引的工作在很多时候往往显得不是很靠谱，并且当我们将数据存储进去后，往往很难进行修改。大量的 Log 文件就需要专门的工作来分析和使用，依赖于人来解析这些日志往往显得不是很靠谱。这时就需要一些重量级的工具，如用 Logstash、ElasticSearch、Kibana 来处理 Nginx 访问日志。

而对那些非专业人员来说，使用 Excel 这样的工具往往显得比较方便。他们不需要操作数据库，也不需要专业知识。只是从某种意义上说，Excel 应该归属于数据库的范畴。

2. 数据库

当我们开始一个 Web 应用的时候（如创建一个用户管理系统），就需要不断对文件进行查询、修改、插入和删除等操作。不仅如此，我们还需要定义数据之间的关系，如这个用户对应这个密码。在一些更复杂的情况下，我们还需要寻找这些用户中对应的一些操作数据等。如果还是将这些工作交给文件来处理，那就是在给自己挖坑。

简单地说，数据库可视为电子化的文件柜——存储电子文件的处所，用户可以对文件中的数据执行新增、截取、更新、删除等操作。

在操作库的时候，我们会使用名为 SQL（结构化查询语言）的领域特定语言来对数据进行操作。

SQL 是高级的非过程化编程语言，它允许用户在高层数据结构上工作。它不要求用户指定对数据的存放方法，也不需要用户了解其具体的数据存放方式。

数据库里存储着大量的数据，在我们对系统建模的时候，也在决定系统的基础模型。对数据库来说，人们有多种不同的选择。

- 选择传统的关系数据库，如常见的 MySQL、SQLite3、PostgreSQL，其特点是建立在关系模型的基础之上。
- NoSQL 是非关系数据库的一类总称，其主要特点是不使用 SQL 作为查询语言，并且其数据格式无固定的模式，常见的有：MongoDB、Redis、Cassandra 等。

由于手动编写 SQL 语句比较烦琐，并且容易出错，因此都会选用 ORM 来辅助开发。ORM 可以实现面向对象编程语言里不同类型系统的数据之间的转换。本书使用了 Django 作为 Web 开发框架，其自带有 ORM，可以支持大部分关系型数据库，如 SQLite、MySQL 等。除此之外，开发者还可以借助 Django-nonrel 等扩展来使用 NoSQL 数据库。

3. 搜索引擎

对以查询为主的网站来说，使用搜索引擎会比数据库更有效。百科上对搜索引擎的定义是如下：

搜索引擎指自动从因特网搜集信息，并经过一定整理以后，提供给用户进行查询的系统。

这样描述并不是非常准确，因为有相当多的网站采用了搜索引擎作为基础的存储服务架构，而且它们并非自动从互联网上搜索信息。搜索引擎的组成应该分成以下三部分。

①索引服务。

②搜索服务。

③索引数据。

索引服务用于将数据存储到索引数据中，而搜索服务正是搜索引擎存在的意义。对于查询条件复杂的网站来说，采用搜索引擎就意味着减少了非常多的烦琐的数据处理事务。在一些架构中，人们用数据库存储数据，并使用工具将数据注入搜索引擎中。

从架构上说，使用搜索引擎的优点是：分离存储、查询部分。从开发上说，它可以让我们更关注于业务本身的价值，而不是去实现这样一个搜索逻辑。

如图 3-3 所示的 Lucene 应用架构，被明显分为以下两部分。

- Index Documents：索引文档部分，将用于存储数据到文件系统中。
- Search Index：搜索部分，用于查询相应的数据。

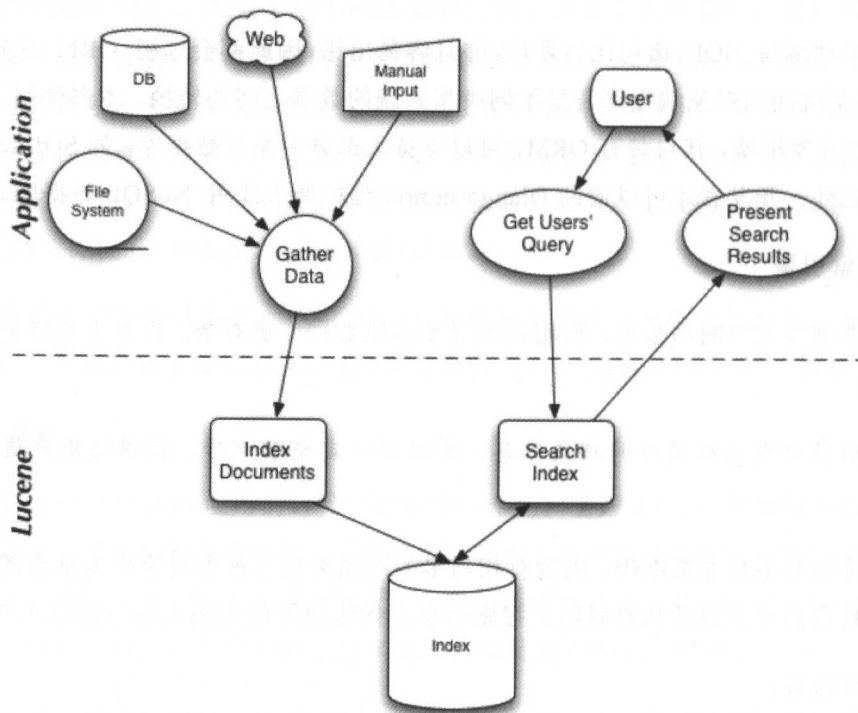


图 3-3 Lucene 应用架构

在实现时，我们只需要编写相应的程序将数据按需要转换，并索引到搜索引擎中。在使用时，则只可以通过搜索接口来查询。

3.1.3 前端选型：UI 框架

事实上，在应用前端技术之前，我们需要搞清楚一件事：是否真的需要一个单页面应用（single-page application, SPA）。单页面是指在一个页面上集成需要的绝大部分功能，它所带来的效果是在运行时类似于桌面应用，又或者是在手机上运行时类似于移动应用。我们将在第9章介绍单页面应用，同时将介绍如何搭建一个纯前端的项目，以及一个基于 Ionic 2 和 Angular 2 的应用。

与后台框架类似的是，前端 UI 框架也分为重量级和轻量级。重量级的框架可以在 UI 上提供更多的组件、更快的速度，但是从定制上就没有轻量级方案方便。轻量级的 UI 框架可以让我们添加需要的功能，并可以在这个基础上开发出自己的 UI 组件库。两者之间很难形成对比，这与 IDE 的选择一样，如果你是新手，建议从具有丰富功能的重量级框架选起。

尽管我们已经使用了 Bootstrap 作为 UI 框架。但是，让我们也了解一下其他一些相当不错的框架。

- **Foundation:** 和 Bootstrap 一样，也是一个组件丰富的 UI 框架，它是由设计公司 zurb.com 推出的开源 CSS 框架。它是主流框架中最早支持移动优先的框架，可用于构建基于任何设备上的 Web 应用。
- **Skeleton:** 是一个超轻量级的前端框架，其代码只有不到 400 行代码，可以基于此来创建自己的 UI 风格。它可以快速地调整出在不同分辨率及设备下的网页显示效果。

除了这些，我们还有诸如 Pure.css 等选择。由于这些简单的介绍无法让读者对此有印象，这里展开讨论。

3.2 Django

综上所述，由于 Django 已经有相当多成熟的 CMS 成功案例，并且更适合于 CMS 的开发，我们将使用 Django 来开发 Web 应用。

3.2.1 Django 简介

Django 最初被开发来用于管理劳伦斯出版集团旗下的一些以新闻内容为主的网站。所以，我们可以发现在使用 Django 的很多网站里，都是作为 CMS（内容管理系统）来使用的。使用 Django 的一些比较知名的网站如图 3-4 所示。



图 3-4 使用 Django 的网站

Django 是一个 MTV 框架，其架构模板看上去与传统的 MVC 架构并没有太大区别。其对比如表 3-1 所示。

表 3-1

传统的 MVC 架构	Django 架构
Model	Model(Data Access Logic)
View	Template(Presentation Logic)
View	View(Business Logic)
Controller	Django itself

在 Django 中, View 只用来描述你要看到的内容, Template 才是最后用于显示的内容。而在 MVC 架构中, 这只相当于是 View 层。它的核心包含下面四部分。

- 一个对象关系映射, 作为数据模型和关系型数据库间的媒介 (Model 层)。
- 一个基于正则表达式的 URL 分发器 (即 MVC 中的 Controller)。
- 一个用于处理 HTTP 请求的系统, 含 Web 模板系统 (View 层)。

其核心框架还包含以下部分。

- 一个轻量级的、独立的 Web 服务器, 只用于开发和测试。
- 一个表单序列化及验证系统, 用于将 HTML 表单转换成适用于数据库存储的数据。
- 一个缓存框架, 并且可以从几种缓存方式中选择。
- 中间件支持, 能对请求处理的各个阶段进行处理。
- 内置的分发系统允许应用程序中的组件采用预定义的信号进行相互间的通信。
- 一个序列化系统, 能够生成或读取采用 XML 或 JSON 表示的 Django 模型实例。
- 一个用于扩展模板引擎能力的系统。

Django 的每一个模块在内部都称为 App, 在每个 App 中都有自己的三层结构, 如图 3-5 所示。

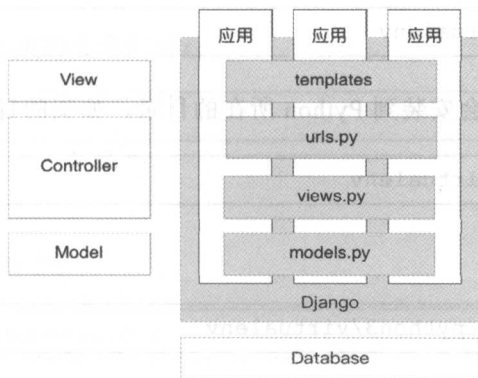


图 3-5 Django 应用架构

这样做不仅可以在开发的时候更容易理解系统，而且可以提高代码的可复用性——因为每一个 App 都是独立的应用，在下次使用时只需要简单复制和粘贴即可。

3.2.2 安装 Django

安装 Django 之前，我们可以用 `virtualenv` 工具来创建一个虚拟的 Python 运行环境。

1. 使用 `virtualenv` 搭建隔离环境

环境问题是一个很复杂的问题，在使用 Python 的过程中，我们会不断安装一些库，而这些库可能会有不同的版本，并且在安装 Python 库的过程中，我们会遇到权限问题，即需要超级用户的权限才能将库安装到系统的环境之下。随后在这个软件的生涯中，还需要保证这个项目所依赖的模块不会发生变动。而这些都是很棘手的一些事，这时就需要创建一个虚拟的运行环境，`virtualenv` 就是这样一个工具。

安装 Python 包需要用到 `pip` 命令，它是 Python 语言中的一个包管理工具。如果你没有安装，可以使用下面的命令来安装：

```
curl https://bootstrap.pypa.io/get-pip.py | python
```

在不同的 Python 环境中，可能需要使用不同的 `pip`，笔者使用的 Python3 的 `pip` 命令 `pip3` 如下：

```
$ pip3 install virtualenv
```

需要注意的是，这将会安装到 Python 所在的目录，如我的目录是：

```
$ /usr/local/bin/virtualenv
```

有的可能会是：

```
$ /usr/local/share/python3/virtualenv
```

在创建这个虚拟环境之前，可以创建一个存储所有 `virtualenv` 的目录：

```
$ mkdir somewhere/virtualenvs
```

现在，我们就可以创建一个新的虚拟环境：

```
$ virtualenv somewhere/virtualenvs/<project-name> --no-site-packages
```

如果想使用不同的 Python 版本，则需要指定 Python 版本的路径。

```
$ virtualenv --distribute -p /usr/local/bin/python3.5 somewhere/virtualenvs/  
<project-name>
```

这里使用的路径名是 py35env，然后将安装相应的 Python 环境：

```
Running virtualenv with interpreter /usr/local/bin/python3.5  
Using base prefix '/usr/local/Cellar/python3/3.5.2/Frameworks/Python.  
framework/Versions/3.5'  
New python executable in /Users/fdhuang/write/py3env/bin/python3.5  
Not overwriting existing python script /Users/fdhuang/write/py3env/bin/  
python (you must use /Users/fdhuang/write/py3env/bin/python3.5)  
Installing setuptools, pip, wheel...done.
```

通过 source 到相应的目录下执行激活就可以使用这个虚拟环境了：

```
$ cd somewhere/virtualenvs/py35env/bin  
$ source activate
```

停止使用只需执行下面的命令即可：

```
$ deactivate
```

2. 安装 Django

安装 Django 的命令如下：

```
$ pip install Django==1.10.2
```

再次会用最新版本的 Django，命令如下：

```
Collecting Django==1.10.2
  Downloading Django-1.10.2-py2.py3-none-any.whl (6.8MB)
    35% |██████████| 2.4MB 32kB/s eta 0:02:16
   100% |██████████| 6.8MB 52kB/s
Installing collected packages: Django
Successfully installed Django-1.10.2
```

下载完后，就会开始安装 Django。安装完后，就可以使用 Django 自带的 `django-admin` 命令。`django-admin` 是 Django 自带的一个管理任务的命令行工具。

通过这个命令，不仅可以用它来创建项目、创建 App、运行服务、数据库迁移，还可以执行各种 SQL 工具等。`django-admin` 的用法如下：

```
$ django-admin <command> [options]
```

下面是 `django-admin` 自带的一些命令：

```
[django]
  check
  compilemessages
  createcachetable
  dbshell
  diffsettings
  dumpdata
  flush
  inspectdb
  loaddata
  makemessages
  makemigrations
  migrate
  runfcgi
  runserver
```

```
shell
sql
sqlall
sqlclear
sqlcustom
sqldropindexes
sqlflush
sqlindexes
sqlinitialdata
sqlmigrate
sqlsequencereset
squashmigrations
startapp
startproject
syncdb
test
testserver
validate
```

现在，让我们来看看这个强大的工具。

3.2.3 创建项目

在 django-admin 自带的命令中，startproject 可以用于创建项目，在这里我们的项目名是 blog，则命令如下：

```
$ django-admin startproject growth_studio
```

这个命令将创建下面的文件内容，而这些是 Django 项目必需的文件。

```
.
├── growth_studio
│   └── growth_studio
```

```
|   |— __init__.py
|   |— settings.py
|   |— urls.py
|   |— wsgi.py
|— manage.py
```

2 directories, 5 files

blog 目录对应的就是 blog 项目，将会放置这个项目的一些相关配置：

①settings.py 包含这个项目的配置。如数据库环境、启用的插件等。

②urls.py 即 URL Dispatcher 的配置，指明了某个 URL 应该指向某个函数来处理。

③wsgi.py 用于部署。WSGI (Python Web Server Gateway Interface, Web 服务器网关接口) 是为 Python 语言定义的 Web 服务器和 Web 应用程序或框架之间的一种简单而通用的接口。

④__init__.py 指明了这是一个 Python 模块。

manage.py 会在每个 Django 项目中自动生成，它可以和 django-admin 做类似的事。如可以用 manage.py 来启动测试环境的服务器：

```
$ python manage.py runserver
```

```
Performing system checks...
```

```
System check identified no issues (0 silenced).
```

```
You have 13 unapplied migration(s). Your project may not work properly until
you apply the migrations for app(s): admin, auth, contenttypes, sessions.
Run 'python manage.py migrate' to apply them.
```

```
October 19, 2016 - 08:29:24
```

```
Django version 1.10.2, using settings 'growth_studio.settings'
```

```
Starting development server at http://127.0.0.1:8000/
```

```
Quit the server with CONTROL-C.
```

```
[19/Oct/2016 08:29:35] "GET / HTTP/1.1" 200 1767
```

```
Not Found: /favicon.ico
```

```
[19/Oct/2016 08:29:35] "GET /favicon.ico HTTP/1.1" 404 1943
```

现在，我们只需要在浏览器中打开 <http://127.0.0.1:8000/>，便可以访问应用程序。

1. Django 后台

Django 很适合开发 CMS 应用的另外一个原因，就是它自带了一个后台管理系统。为了启用这个后台管理系统，我们需要配置数据库，并创建相应的超级用户。

在访问后台之前，需要选择一个数据库，并安装相应的数据库。Django 内建支持下面的一些数据库：

```
'django.db.backends.postgresql_psycopg2'
```

```
'django.db.backends.mysql'
```

```
'django.db.backends.sqlite3'
```

```
'django.db.backends.oracle'
```

这里将使用 SQLite 3 作为我们的数据库，因此，需要在计算机上安装 SQLite 3 数据库。

几乎所有的 GNU/Linux 操作系统都带有 SQLite；最新版本的 Mac OS X 会预安装 SQLite；Windows 用户可以使用在第 1 章提到的包管理工具下载，或者可以直接到 SQLite 官网下载，下载地址为：<https://sqlite.org/download.html>。

安装完后，要确保可以使用命令 `sqlite3` 来访问数据库：

```
$ sqlite3
```

```
SQLite version 3.9.2 2015-11-02 18:31:45
```

```
Enter ".help" for usage hints.
```

```
Connected to a transient in-memory database.
```

```
Use ".open FILENAME" to reopen on a persistent database.
```

```
sqlite>
```

Django 默认使用的是 SQLite 3，如下所示的命令是 settings.py 中默认的数据库配置：

```
# Database
# https://docs.djangoproject.com/en/1.10/ref/settings/#databases

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
    }
}
```

上面的配置中使用的是 SQLite3 作为数据库，并使用了当前目录下的 db.sqlite3 作为数据库文件。

如果想使用其他数据库，可以在网上寻找解决方案，如用于支持使用 MongoDB 的 django-nonrel 项目。不同的数据库有不同的配置，如下命令是使用 PostgreSQL 的配置。

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql_psycopg2',
        'NAME': 'mydatabase',
        'USER': 'mydatabaseuser',
        'PASSWORD': 'mypassword',
        'HOST': '127.0.0.1',
        'PORT': '5432',
    }
}
```

接下来可以运行数据库迁移，只需要运行相应的脚本即可：

```
$ python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, sessions
Running migrations:
```

```
Applying contenttypes.0001_initial... OK
Applying auth.0001_initial... OK
Applying admin.0001_initial... OK
Applying admin.0002_logentry_remove_auto_add... OK
Applying contenttypes.0002_remove_content_type_name... OK
Applying auth.0002_alter_permission_name_max_length... OK
Applying auth.0003_alter_user_email_max_length... OK
Applying auth.0004_alter_user_username_opts... OK
Applying auth.0005_alter_user_last_login_null... OK
Applying auth.0006_require_contenttypes_0002... OK
Applying auth.0007_alter_validators_add_error_messages... OK
Applying auth.0008_alter_user_username_max_length... OK
Applying sessions.0001_initial... OK
(growth-django)
```

在上面的命令中，我们会创建相应的数据库模型，并依据迁移脚本来创建相应的数据，如默认的配置等。

最后，可以创建一个相应的超级用户来登录后台。

```
$ python manage.py createsuperuser
Username (leave blank to use 'fdhuang'): phodal
Email address: h@phodal.com
Password:
Password (again):
Superuser created successfully.
```

输入相应的用户名和密码，即可完成创建。然后访问 <http://127.0.0.1:8000/admin>，输入上面的用户名和密码就可以进入后台，如图 3-6 所示。

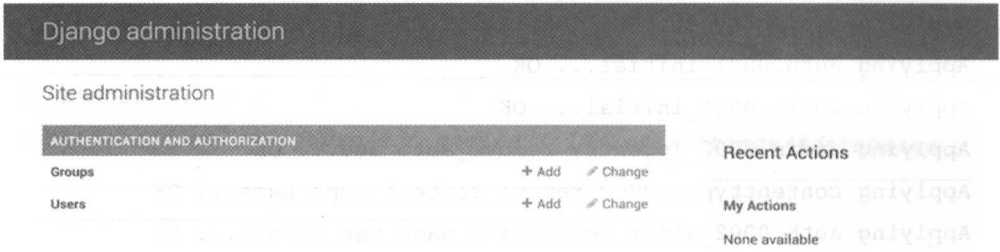


图 3-6 Django 后台

如果你对英文介绍不是很熟悉，可以进行 `l18n` 设置，修改 `LANGUAGE_CODE = 'zh_hans'`，就可以将语言设置为简体中文，如图 3-7 所示。

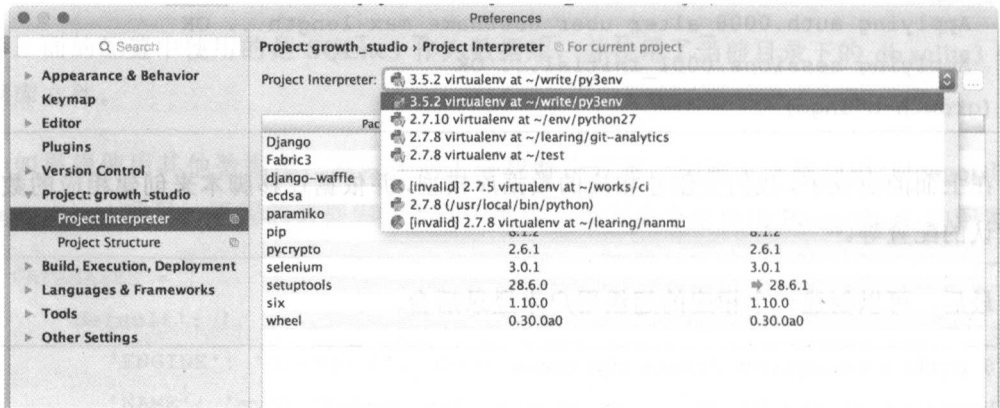


图 3-7 设置 PyCharm

2. 提代码

在创建完应用后，我们就可以进行第一次提交，这样的提交信息（commit message）通常是 `init project`。如果在那之前，你没有执行 `git init` 来初始化 `git`，就需要执行这个命令。

需要注意的是，数据库文件不应该添加到项目里，为了避免手误产生一些问题，我们需要添加一个名为 `.gitignore` 的文件用于将一些文件名加入忽略名单，如下命令是常用的 `python` 项目的 `.gitignore` 文件中的内容：

```
*.pyc
*.db
```

```

*.sqlite3

# Byte-compiled / optimized / DLL files
__pycache__ /
*.py[cod]
*$py.class

```

当我们添加完这个文件后，git 就会识别这个文件，并忽略原来不需要添加到源码库里的文件。

3.3 从真实世界到代码

在我毕业一年后，由于项目缺人，我开始承担 Tech Lead 的责任，主导对新系统的设计，开始学习架构方面的知识，并接触 DDD 的思想。随后，再回到一个普通开发人员的角色。从编码到架构，再回到实际的编码中，对这些概念就有了更深刻的印象。

我们所写的代码在某种程度上都反映了真实世界的模型、行为等。一个比较常见的模型就是购物模型。同时，这也是一个很好的展示前后端分离的模型，如图 3-8 所示。

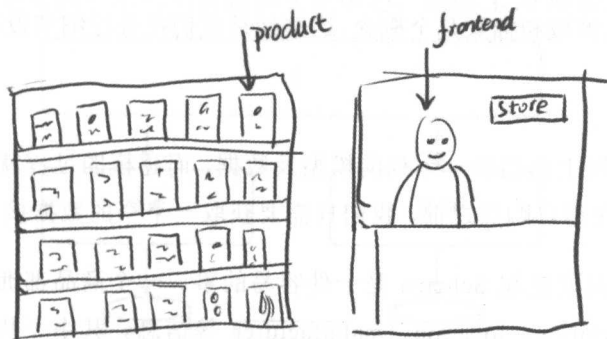


图 3-8 便利店前后台模型

对于一般的便利店来说，只有一个售货员，他负责整个商店的一系列事务。从某种意义上说，售货员就是整个系统的核心，负责系统的业务和事件。

一般来说，在一个购买流程里会有三个主要的人或物。

- 售货员：一般来说，售货员只会在最后的结账流程中以及顾客询问时做出响应。
- 货物：就是一堆模型。
- 顾客：浏览商店、对比商店等。

如果要构建这样一个系统，只需要区分出系统的各个部分，那么剩下的事情就变得很简单了。这样一个系统仍然是相当复杂的，我们还需要考虑到库存、发票、产品清单等子系统。因此，这里只关注于用户购买的过程。

3.3.1 模型、领域、抽象

从购买过程来说，顾客所要做的事情就是：

- 浏览、对比商品。
- 加到购物车。
- 结账、付钱。

对应的有模型、领域和抽象几个概念。这些都是我们在进行细节设计时需要具备的能力。

1. 模型

这些商品在实现上相当于一系列的模型及数据，而建模的过程就是在建立业务需求与模型之间的关系。在用户购买之前，我们只需要获取一个个的数据接口，并展示这些数据。

对应于这些商品要建起 Schema 是一件容易的事。一个商品都拥有一些共同的元素：price、name、description、location、manufacturer 等信息。其中一些属性还会有复杂的对应关系，如图 3-9 所示。

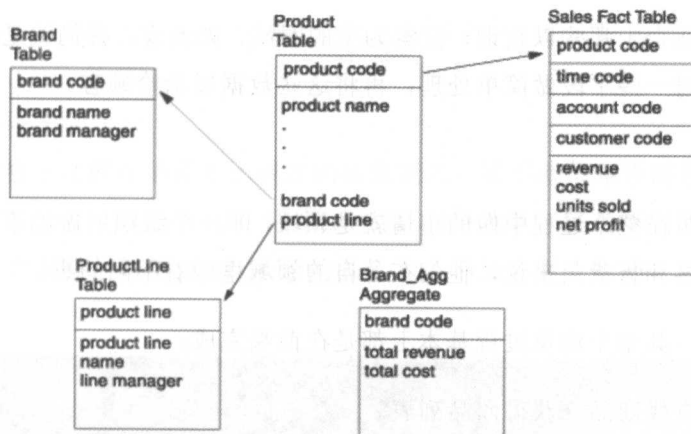


图 3-9 Product Schema

这些需要在建立数据库的时候，尽可能地明确好它们之间的关系。由于业务本身是难以预料的，你可能和我们之前的项目一样需要一个 additionInfo 字段，来用 JSON 存储一些额外的字段。当然，如果你使用的是 NoSQL 就更好了。

你最好还使用了读写分离架构，一种比较常见的用法就是 CMS 网站，人们使用数据库来存储内容，使用静态页面来展示这些内容。比较好的实践还有 CQRS（命令查询职责分离模式），用于 CRUD（增、删、改，当然也可以查）的 Command，以及 Query 的查询分开。简单地说，就是有两个不同的数据持久化中心，如图 3-10 所示。

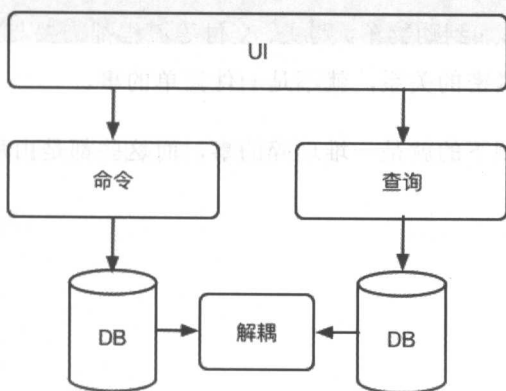


图 3-10 Basic CQRS

这一点特别适合于那些以查询、搜索为主的网站，如淘宝。我们就是在这里提供了数据库的数据，并对一些字段做简单处理，再将这些数据展示给顾客。

2. 领域

顾客和售货员在整个过程中做的事情就是领域，即一个组织所做的事情以及其中所包含的一切。对顾客和售货员来说，他们在各自的领域里做着不同的事。

对顾客来说，其整个浏览过程基本上都是在前端完成：

- 搜索、查找商品→获得商品列表。
- 查找商品详情。
- 切换到下一个商品。

在这个场景下就特别适合于上面说到的读写分离架构。在浏览过程中，对用户的数据进行监控，以用于了解用户的行为，改善用户体验。这也是很常见的功能，或者说它们是无处不在的模式：

- 结果页 / 列表页。
- 详情页。

随后的用户收藏、添加到购物车、购买、交付等流程都需要与后台进行更紧密的交付。而这些都和售货员都有紧密的关系，就不是一件简单的事。

用户购买完成后，剩下的就是一堆琐碎的事，而这些都是由后端来完成的：

- 订单子系统。
- 物流系统。
- 发票系统。
- 支付系统。

对用户来说，一种最简单的情况就是亚马逊，你只需要单击“一键下单”即可。不需

要关心后面的操作，同样，这也适合于我们的业务场景——顾客只需要付钱即可。

3. 抽象

总的来说，整个过程还是需要比较好的抽象能力，要不就很难弄清其中的过程了。抽象是很神奇的东西，也可以分为几个不同的境界，简单地说，就是不同的人看上去就有不同的东西。如有的人看到如图 3-11 所示的画认为还不如小学生画的，但有的人就会惊呼大师之作。

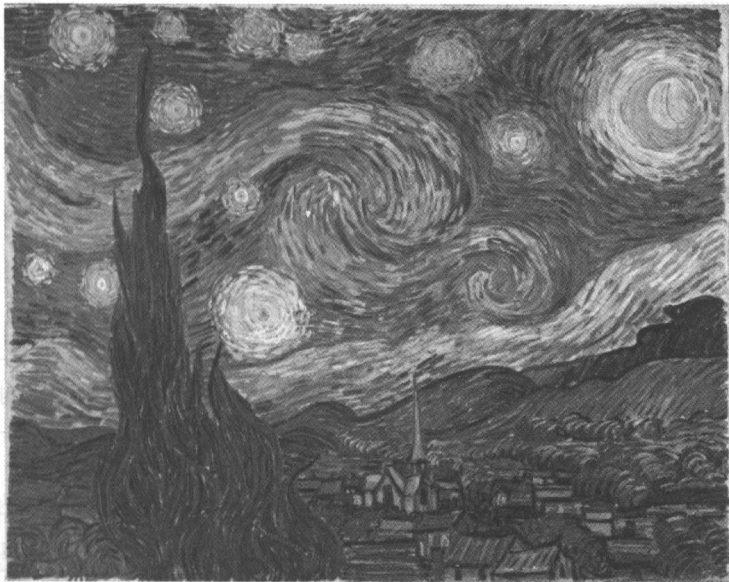


图 3-11 星空

这种想法类似于最初我对设计模型的理解：

- 一开始不以为然。
- 然后发现很棒。
- 接着使用过度。
- 最后再理解到简单的应用上。

这些都在随着编程生涯的展开而发生一些变化，我们不断地抽象出一些概念，以至于到了最后刚进入这个行业的人都看不懂。但是，这些都是一点点在一层层抽象的基础上产生的。在讨论软件领域的抽象时，我们会讨论设计模式、API 设计等一些抽象的概念。而这些都依赖于我们有足够的编程知识和足够的理论知识作为支撑。

再让我们回到前后台的设计上去。

3.3.2 前后端分离

货物和售货员之间有着很明显的前后台关系。下面将它们分为前后端来讨论。

1. 后台

典型的 Web 应用框架类似如图 3-12 所示的架构。

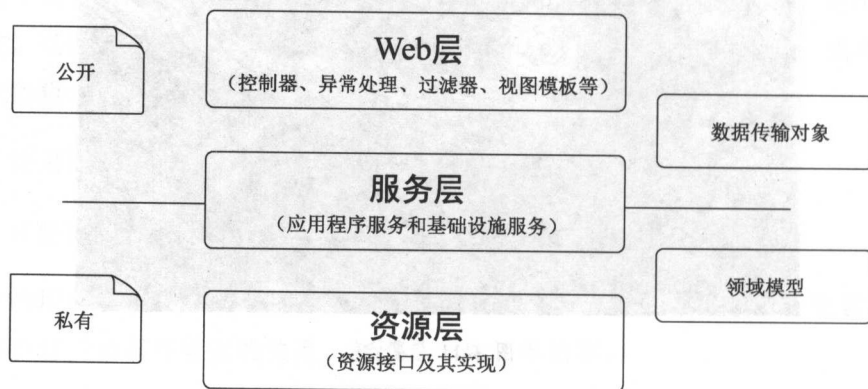


图 3-12 Spring Web App Architecture

让我们继续上面的步骤来完成购买流程，后台除了提高上面的商品信息以外，在购买的时候还需要对用户进行认证、授权。当然，注册就是另外一个话题了。而这些是基于我们不信任该网站的情况下，如果我们相信这个网站的服务，那么我们完成直接由微信或者支付宝在扫描完支付二维码，写上的收货地址，就可以买下这个产品。

所有的这些都可以向前台提供对应的 API。理想情况下，对应于不同的模块可以有不同的服务，如图 3-13 所示。

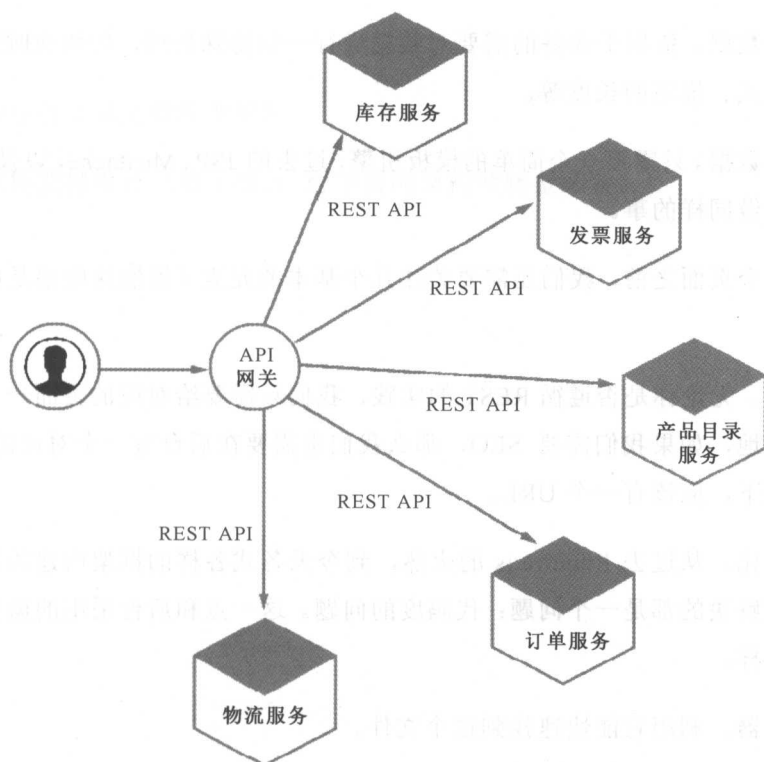


图 3-13 MicroServices

但是现实并不总是这么美好的，当前情况下则可以——毕竟所有的用户都应该能浏览所有的商品，这时就不需要做特殊的处理。在这个过程中，我们还有一系列的操作需要在后台完成。而复杂的过程实际上还存在于前端的逻辑中。

2. 前端

开始时，我们需要这样做去获取一个个商品详情。这些数据也可以在返回页面模板的时候直接将 API 数据填充到 HTML 中——带后台渲染的 React 都是这样做的。然后在用户浏览的过程中还需要遵循以下数据流程。

- 获取数据。无论是 Ajax 还是新的 Fetch API，都可以做这样的事，从后台获取需要的数据。

- 处理数据。依据于业务的需要对数据进行一些特殊处理，如修改时间、价格的显示格式、描述的长度等。
- 显示数据。只需要一个简单的模板引擎，过去的 JSP、Mustache，以及今天的 React 都在做同样的事。

在进入这个页面之前，我们还需要关注几个基本的元素（虽然这些都是由框架来解决的）。

- 路由。无论你是否遵循 REST 的实践，我们只需要给对应的页面一个 URL 即可。相应地，如果我们需要 SEO，那么我们也需要在后台有一个对应的 URL。理想情况下，应该有一个 URL。
- 模块化。从过去 Require.js 的火热，到今天各式各样的框架内建的模块化框架，它们解决的都是一个问题：代码度的问题。这一点和后台采用的微服务架构的缘由一样。
- 控制器。利用它能快速找到这个文件。

即使我们的应用不是一个单页面应用，上面的因素仍然是存在的。作为前端，我们不仅要负责页面的美观，还要对用户的事件做出响应，并且做好用户体验。

最后，当用户买下东西的时候，也需要这样的交互流程。

3.4 小结

在本章，我们简单介绍了如何对架构进行设计，以及如何进行技术造型，并详细介绍了不同语言的后端框架、UI 框架，同时结合在第 2 章中实现的着陆页，完成了项目的基本框架。最后，还介绍了如何将真实世界的业务转换为代码。

在下一章，我们将介绍如何为 Web 应用设计构建流，并为 Web 应用打造一个构建完整的系统。

参考书目

- 《程序员必读之软件架构》
- 《软件架构设计（第2版）：程序员向架构师转型必备》

第 4 章

构建系统及其 workflow

在本章中，我们将搭建本书的基础构建系统，并详细介绍每一个子模块的功能，以及这样做的重要性。

xxx: 你猜猜我们现在应该做些什么?

Phodal: 难道不是开始写代码吗?

xxx: 你平时都是这样做的? 不需要 xx, 也不需要运行测试?

Phodal: 这个嘛……

xxx: 在开始写代码之前, 需要编写一些简单的轮子来辅助开发。这些简单的代码可以帮我们运行起服务、测试、打包上传等。

Phodal: 现在需要我自己做这个轮子?

xxx: 嗯, 是的。优秀的程序员与普通的程序员在效率上存在着 10 倍的差距, 而有些差距是可以通过创建这些轮子来缩短的。

当我们从头开始创建一个项目时, 就需要搭配好一个构建系统来改善代码质量, 同时可以节省更多的时间来解放生产力。我们可以使用测试和代码风格检测来提高代码质量, 也可以用自动化打包来提高部署流程。而这些都可以通过搭建好构建系统来提高。

4.1 构建流

构建系统是用来从源代码生成用户可以使用的目标的自动化工具。这些目标可以包括库、可执行文件或者生成的脚本等。

而构建工具是构建系统的基础元素, 它可以用于执行一系列的有序任务来实现这个过程的自动化。不同的语言拥有不同的构建工具, 当然这些工具也可以用在不同的语言上, 常用的构建工具有: 用于 C 语言的 GNU Make、CMake; 用于 Java 语言的 Apache Ant、Gradle; 用于 Node.js 的 NPM、Gulp、Grunt, 或者最新的 Yarn 等。此外, 所有的集成开发环境比如 Qt Creator、Microsoft Visual Studio 等都对它们支持的语言添加了自己的构建系统配置工具。

使用纯粹的构建工具来实现一个构建系统，则可以使整个项目的构建实现全面的自动化编译、测试、打包部署。现在，让我们借助于一个现有的构建系统的流程来了解构建系统的构建流程。

如图 4-1 所示的系统是一个前端为单页面应用，后端使用 Java 及 Spring MVC 提供 RESTful API 的前后端分离项目的编译流程。从左到右开始，分为两部分：后台部分、前端以及仿造后台 API 的 Mock 服务。由于这个项目在组成上分为前后端两个不同的团队，因而系统在构建时分为前端及后台两部分。

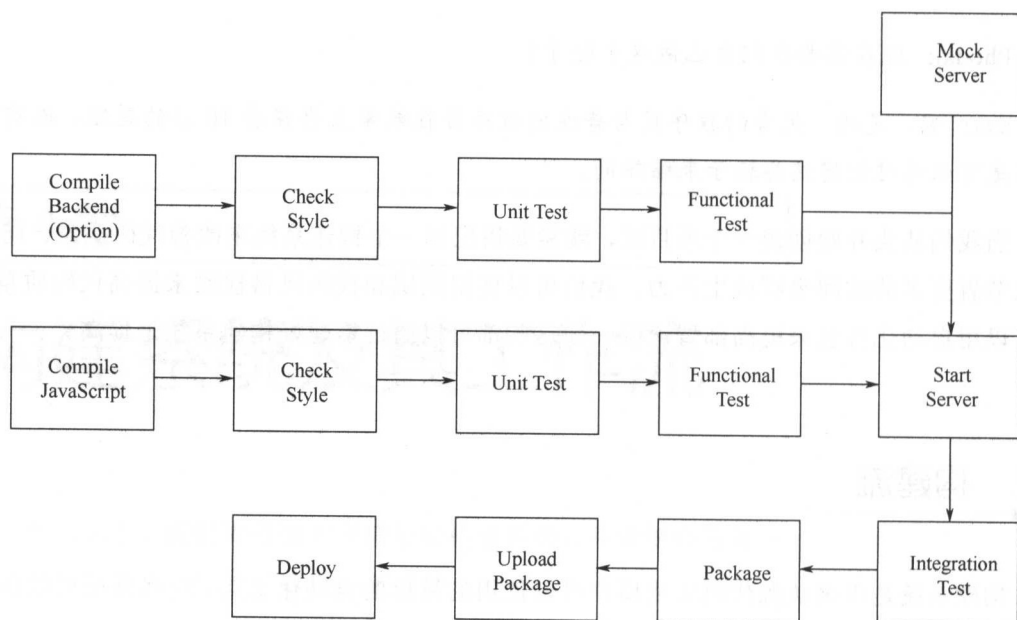


图 4-1 构建过程

在进行前端构建时，我们将会对 JavaScript 文件中的环境变量进行处理，然后进行代码风格检测（Check Style）来保证代码中的风格是一致的。接着，将运行对应的单元测试及功能测试。最后在这部分将启动一个 Mock 服务来进行前端的集成测试。

在进行后端构建时，我们也将进行类似的流程，代码风格检测、编译代码、单元测试、功能测试等。最后会进行集成测试来保证这个 API 返回结果和我们需要的是一致的。

在完成上述两部分的流程后，将打包代码，并上传代码到包中心，最后将代码部署到服务器上。而这一个个步骤正是我们在构建过程中所需要的：

- **编译**。对那些不是用浏览器的前端项目来说，如 ES6、CoffeeScript，它们还需要将代码编译成浏览器使用的 JavaScript 版本。对 Java 语言来说，它需要一个编译的过程，在这个编译过程中，会检查一些语法问题。
- **代码风格检测 (Check Style)**。通常，我们会在项目里定义一些代码规范，如 JavaScript 中使用两个空格的缩进，在 Java 的 Checkstyle 中一个函数不能超过 30 行的限制。
- **单元测试**。作为测试中最基础也是最快的测试，这个测试将集中于测试单个函数是否正确。
- **功能测试**。功能测试的意义在于，保证一个功能依赖的几个函数组合在一起也是可以工作的。
- **Mock Server**。当我们的代码依赖于第三方服务的时候，就需要一个 Mock Server 来保证功能代码可以被独立测试。
- **集成测试**。这一步将集成前台、后台，并且运行起最后将上线的应用。接着依据用户所需要的功能来编写相应的测试，以保证每个功能是可以工作的。
- **打包**。对部署来说，直接安装一个 RPM 包或者 DEB 包是最方便的事。在这个包里会包含应用程序所需的所有二进制文件、数据和配置文件等。
- **上传包**。在完成打包后，我们就可以上传这个软件包到某个包管理中心。
- **部署**。在完成上面的步骤后，就可以在线上环境从包管理中心获取这个包，安装所需要的依赖，最后再安装这个软件包。

在不同的环境下运行时，它们又有一些细微的区别。在开发环境下，我们主要关注如何更方便地开发和测试以及如何提高代码质量。在生产环境下，我们关注如何自动化部署流程。

4.1.1 搭建开发环境

在搭建开发环境的构建系统时，需要关注以下两点。

- 提高效率，对于大部分事务的自动化，如自动编译代码、自动重启服务。
- 代码质量，编码完成时，应转而关注代码质量。

下面详细介绍这两点内容。

1. 提高效率

(1) 依赖管理

我们在第 1 章时提到了包管理的一些基础知识，对一个成熟的编程语言说，也搭配有对应的包管理工具。只是这里的包管理工具不仅具有包管理的功能，它还会负责对包所依赖的子依赖进行管理。如我们在安装 Vim 软件的时候，会依赖于 vim-runtime，这里包管理工具所自带的依赖管理功能将会自动安装这个依赖包。

对应的也会有相应的版本管理机制，如在第 3 章里，我们使用 `pip install Django==1.10.2` 来指定安装的 Django 包的版本，这些配置也会也在相应的配置文件里。在一些语言里，其包管理工具可以指定包依赖的子依赖的版本，如可以限定 vim-runtime 的版本为 xx，如 Ruby 语言里的 Gemfile.lock，又或者是 Nodejs 里的 shrinkwrap。

除了在依赖管理配置里写上所需要使用的库的版本，还可以在这个文件里指定包的中心。在中大型的软件公司里，由于安全性及便利性的原因，他们都会建立自己的包中心（又或者称为仓库中心）。在这些仓库中心里，将会同步不同语言的仓库中心的包，并且也会将自己开发的软件包上传到这个中心，以便组织里的其他开发者使用。

(2) 自动重载

在早期的 Web 开发中，修改完后台代码时，需要手动重启后台服务；修改完前端代码时，需要手动刷新前端界面。而这些步骤事实上都可以交由自动化工具来完成，在后台开发时，这部分需要依赖于框架来实现，又或者是构建工具。而在前端开发时，则可以手

动创建这样的服务。当检测到本地代码修改时，则自动编译代码，再重新加载前端页面。

当我们开发前端项目时，可能会用到 TypeScript 或者 ES6。为了在浏览器上运行它们，需要将其转换为 JavaScript，因此，每当我们修改代码时，就需要做一次转换。又或者是使用 SASS 或者 LESS 作为样式开发语言时，就需要在运行时将其转换为 CSS。而转换为 JavaScript 和转换为 CSS 这两点都是在自动重载时所需要做的。

2. 代码质量

(1) lint 检测编程风格

当 C 还是一门新型的编程语言时，还存在一些未被原始编译器捕获的常见错误，所以程序员们开发了一个被称为 lint 的配套项目用来扫描源文件，并查找问题。对应于不同的语言都会有不同的 lint 工具，在 Python 语言中有 Pylint，在 JavaScript 中就有 JSLint，在 Java 里有 lint4j 等。我们使用这类工具来帮助我们从基本的语法上提高代码质量，比如：

- 变量定义规范。在不同的语言里存在不同的变量使用方法。如在 Python 里，不使用诸如 GetCurrentPage 类似的驼峰命名方式，转而使用类似于 get_current_page 的下划线方式。当我们使用驼峰来取变量时，所使用的 Lint 工具就会提示这个错误。
- 代码格式规范。不同的人、团队、语言对编程风格有偏好，而在团队合作时，需要保持这些风格的一致性。诸如左括号是否需要换行，右括号后是否需要换行，单个 if 语句是否需要括号等，我们都需要对其有合理的规范，一种比较理想的方法是参考语言官方推荐的编码风格。
- 限制语言特性。由于历史原因或者语言本身的语法糖，语言本身会有一些怪异的特性，可以用来实现一些特殊的功能，诸如 JavaScript 中的 eval 可以对字符串进行求值，而这些特性有可能是危险的，又或者是难以掌控的。因此，需要对这类特性进行限制，仅在不得已的情况下才考虑使用它。
- 代码行数限制。这是一种简单、粗暴的降低代码复杂度的方式，可以将单个函数的代码行数限制在 30 行以内，超过时就得提取出一个新函数。当提取出新的函数时，也就意味着看不到长长的函数。

- 多重嵌套限制。随着业务的扩展或者对异常的处理，我们很有可能在一个 if 里嵌入一个 if 语句，然后又嵌入一个 if 语句，直至代码变得越来越难以阅读。因此，对这种代码的多重嵌套可以限制其为 3 层。当超过 3 层时，就需要对其进行重构。

另外，还有一些值得考虑的限制，诸如：

- 未用到的代码。
- 代码拥有注释。
- 函数里没有相应的文档。

值得注意的是，这样做的主要原因是保持团队的代码风格一致。即使我们已经做出了一套规范，如果没有对应的代码检视机制，还是需要依靠工具来帮助我们提高代码质量。

(2) 运行测试

在准备提交代码到服务器的时候，需要运行测试来保证不破坏系统原来的功能。当然，我们也需要在这时添加对现有代码的测试，运行单元测试、功能测试、集成测试等。

对单元测试和功能测试而言，我们可以使用测试框架自带的测试命令来运行，只需要调用测试框架的接口即可。

对集成测试来说则稍微复杂一些，我们需要运行起一个真实的服务器，使用基于 Selenium 或者类似的自动化测试工具来在浏览器上对页面进行操作，或者是在前后端分离的项目里运行一个仿造服务器，再运行前端项目，最后对页面进行测试。

4.1.2 准备生产环境

在编写构建脚本的时候，我们还需要考虑如何利用构建系统来实现自动化部署。通常来说，这些脚本并不会在本地运行，而是运行在持续集成服务器上。我们将在第 8 章详细介绍。当实现了对代码从打包到部署的自动化后，就可以实现从提交代码到服务器，以及自动化部署代码到测试机器上。

为了实现自动化部署，我们还需要在构建脚本里添加运行环境支持和打包及发布。

1. 运行环境

在服务器上部署网站的时候，需要一些基础软件，如 HTTP 服务器、数据库软件、语言运行环境。因此，我们需要在服务器上安装这些软件，并修改一些配置。而这些都是依赖于脚本来实现的，既可以编写安装脚本在服务器上运行，又可以使用虚拟环境来完成这些工作。除此之外，我们还需要准备好生产环境的依赖配置等。

随后，只需要安装软件包，并重新服务即可。在必要的时候需要考虑平滑升级的方案。

2. 打包及发布

最简单的打包方式可以是直接压缩源码包，并上传至某个包管理中心。接着在服务器上只需要下载这个软件包，并执行相应的安装脚本即可。除此之外，我们也可以考虑在持续集成服务器上编译好代码，并面向服务器所使用的操作系统打包对应的软件包，便可以在服务器上直接安装。在 GNU/Linux 系统的软件包里通过包含已压缩的软件文件集以及该软件的内容信息，常见的软件包有 DEB、RPM、压缩文档 tar.gz。通常来说，这些包会含有下列信息：

- 环境搭建脚本。即在目标服务器安装所需要的软件包、修改配置、创建用户等。
- 依赖关系说明。
- 软件包。即在之前步骤里生成的最后的软件包，如 Java 里的 jar 包，或者是软件的相关版本的源码。
- 部署脚本。在运行软件之前，需要复制相关的软件包到相应的目录、安装依赖、重启 HTTP 服务器等一系列操作。

我们需要做的就是将这些信息和操作放入一个打包配置文件里，如 RPM 中的 spec 文件。首先，需要在这个文件里定义好包名、概述信息、版本号、开发者等基本信息。随后在不同的阶段里执行不同的安装脚本，如在 %build 配置段里执行构建指令，%install 配置段里执行安装命令，在 %files 配置段里执行一些配置操作等。最后，再执行 rpm build，就

可以构建生成可执行文件。

上传包之前需要创建一个相应的文件服务器或者相应的软件源，并且对产品环境的服务器来说，我们还需要指定软件源才能安装这个包。以 Ubuntu 为例，Ubuntu 里的许多应用程序软件包是放在网络的服务器上，这些服务器网站被称为“源”，从源里可以很方便地获取软件包。

4.2 打造后端构建系统

当着手打造一个项目的构建系统时，我们需要考虑选择一个合适的构建工具，并开发设计项目的构建流程。尽管不同的语言有流程上的一些差异，但总的来说，其流程还是相似的。这个构建工具算得上是一个流程工具，可以创建不同的任务、执行系统命令、按顺序执行不同的任务等。在一些语言里，只有一个构建工具，因此，并没有太多的可选择性。这里只需要创建好构建流程。

基于现在的需求，我们可以设计出如图 4-2 所示的流程。

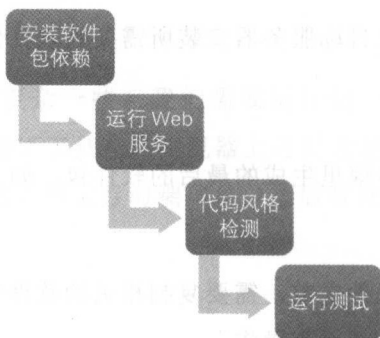


图 4-2 构建流程

这里的构建流程只是开发环境所需要的流程而设计的。对部署以及持续集成环境的构建流程则需要重新设计。

- 安装软件包依赖。

- 运行 Web 服务。
- 执行代码风格检测。
- 运行测试。

除此之外，我们还可以根据上面的几个步骤来创建一个预提交的任务。为了方便与第6章使用的技术栈结合，我们将使用 Fabric 来创建构建系统。

4.2.1 使用 Fabric 搭构建系统

Fabric 是一个基于 Python 的命令行部署工具。它可以用流水线化的方式执行 SSH 以部署应用或者其他系统管理任务，并且可以执行 Shell 命令。我们将借助执行 Shell 命令这一点来搭建本地的构建系统，只需要执行对应的 Shell 或者脚本来实现开发环境的自动化。

在开发之前，需要先安装 Fabric。由于 Fabric 官方还未提供对 Python 3 的支持，因此，需要第三方提供的 Fabric Python 3 的库。

```
pip install fabric3
```

如果安装时没指定好 fabric 的版本，就会遇到下面的错误：

```
Traceback (most recent call last):
```

```
File "/Users/fdhuang/write/py3env/bin/fab", line 7, in <module>
```

```
    from fabric.main import main
```

```
File
```

```
"/Users/fdhuang/write/py3env/lib/python3.5/site-packages/fabric/main.py",  
line 13, in <module>
```

```
    from operator import isMappingType
```

```
ImportError: cannot import name 'isMappingType'
```

这个结果表明当前使用的 Python 版本是不对的，在使用的时候记得检查当前的 Python 版本。

按照国际惯例，我们先来创建一个简单的“hello world”了解大概内容。在项目的目录下创建一个 `fabfile.py` 文件，并输入下面的内容：

```
def hello():  
    print("Hello world!")
```

安装完 `Farbic` 时，它会顺带安装一个名为 `fab` 的工具，我们直接借助这个工具来执行下面的命令：

```
$ fab hello  
Hello world!
```

```
Done.
```

这个 `fab` 命令将来从 `fabfile` 文件里寻找对应的函数来执行。因此，实际上是在将流水线以代码的方式实现。

现在可以从最基本的一步开始：安装软件包依赖。

1. 软件包依赖安装

Python 语言里并没有规定统一的包管理形式，通常使用 `requirements.txt` 作为配置文件来管理软件包的依赖。在第 3 章中使用下面的命令来安装 `Django`。

```
$ pip install Django==1.10.2
```

现在只需要在项目的目录下创建一个 `requirements.txt` 文件，并将 `Django==1.10.2` 添加到文件中即可。随后，就可以执行下面的命令来安装这些依赖包：

```
pip install -r requirements.txt
```

这里的 `-r` 参数指定了从某个文件来进行依赖管理，也就是这里的 `requirements.txt`。而当我们有多个不同的环境需要配置时，就会创建一个 `requirements` 文件夹。在这个文件夹里放置不同环境下的配置，如 `dev.txt` 用于开发环境，`prod.txt` 用于生产环境。

在 Fabric 里有两种不同的运行方式，一种是在本地执行命令，另一种则是在远程机器上。这里将借助 `local` 函数来执行本地的安装命令：

```
from fabric.api import local

def install():
    local("pip install -r requirements.txt")
```

实际上，我们做的只是将命令放到 `install` 函数里来执行罢了。在构建系统里依据需求创建函数，最后将这些函数汇集到一起就可以实现我们的功能。

接着，可以在这个方法里添加一个参数以在不同的环境（即开发和生产环境）里安装不同的依赖：

```
from fabric.api import local

def install(requirements_env="dev"):
    local("pip install -r requirements/%s.txt" % requirements_env)
```

现在，只需要执行下面的命令，就可以安装 `dev` 环境的配置：

```
$ fab install
```

在 `install` 方法里，默认将使用 `requirements` 目录下的 `dev.txt`。当传入一个参数时，如 `prod`：

```
$ fab install:prod
```

那么 `install` 方法中的 `requirements_env` 将赋值为 `prod`，我们就执行 `pip install -r requirements/prod.txt`。现在要做的就是创建不同环境下的包依赖管理文件。

现在，我们完成了简单的前期环境准备，接着创建一些能帮助我们快速开发的任务。

2. 运行 Web 服务

现在可以手动替换常用的一些命令，比如手动运行服务：

```
from fabric.api import local

def runserver():
    local("./manage.py runserver")
```

上面的代码可以将平时经常输入的 `python manage.py runserver` 简化成 `fab runserver`，也可以取一个更简单的函数名，如 `run()`。

以此类推，我们可以写出其他相似的命令来简化开发。可以用 `list` 参数列出已有的任务，代码如下：

```
fab --list
```

这时它会列出文件中所有的方法：

```
Available commands:
```

```
install      Install requirements packages
runserver    Run Server
```

在 `install` 后面的 `Install requirements packages` 表明了这个方法的用途，通常在方法后面使用文档注释符（即三个双引号，`"""`）来显示在命令行结果上。

随着功能的增多，我们会在 `fabfile` 里写上相当多的方法，这些方法有时并不希望在罗列用法的时候显示出来。这时就需要使用到 `@task` 装饰器，只需要在那些想暴露到外部调用的函数里使用这个装饰器即可。如下面的代码：

```
from fabric.api import local
from fabric.decorators import task
```

```
@task
def runserver():
    """Run Server"""
    local("./manage.py runserver")
```

在方法 `runserver` 的前面添加了 `@task`，这时当重新列出可用命令的时候，它就只会显示那些带有 `@task` 装饰器的函数。

3. 代码风格检测

对于使用四个空格还是使用 `Tab` 键进行缩进，不同语言有着不同的差异，对于不同的开发者写出来的代码风格都有所差异。如同 Python 语言的创建者 Guido van Rossum 说的一样：代码更多的是用来读而不是写。因此，Guido 在很早的时候写了一份 *Python Style Guide*，后来在 Python 语言开发者基于这份指南制定了一份主 Python 版本标准库的编码约定——*PEP 8—Style Guide for Python Code*，简称为 `pep8`。

为了更好地遵循这个风格，一个名为 Johann C. Rocholl 的开发人员基于此写了一个名为 `pep8` 的工具来检测代码风格。同样，我们只需要使用 `pip` 来安装这个工具：

```
pip install pep8==1.7.0
```

安装完后，就可以对代码进行风格检测：

```
$ pep .
./growth_studio/settings.py:89:80: E501 line too long (91 > 79 characters)
./growth_studio/settings.py:92:80: E501 line too long (81 > 79 characters)
./growth_studio/settings.py:95:80: E501 line too long (82 > 79 characters)
./growth_studio/settings.py:98:80: E501 line too long (83 > 79 characters)
```

从上述代码可知，在用 Django 生成的 `settings.py` 文件里有几行代码太长，即大于 80 个字符。而我们的代码是：

```
AUTH_PASSWORD_VALIDATORS = [
    {
```

```

        'NAME':
        "django.contrib.auth.password_validation.UserAttributeSimilarityValidato
        r",
        },
        ...
    ]

```

这本身是一个配置选项，只是这个选项对应的是一个类。然后就变成了一个非常有趣的问题：`pep8` 本身是为了更好的可读性而推荐使用的。如果对这个类进行换行，将会失去这个可读性。因此，这份指南在最开始的地方提到了以下几点可以违反指南：

- 遵循指南会降低可读性。
- 与周围其他代码不一致。
- 代码写在引入指南之前，暂时没有理由修改。
- 旧版本兼容。

因此，可以先忽略掉这部分警告。再依据上面的步骤来创建一个任务：

```

@task
def pep8():
    """ Check the project for PEP8 compliance using `pep8` """
    local('pep8 .')

```

如果你是一个完美主义者，可以考虑使用 `Pylint` 来写出更符合规范的代码。如下报告是使用 `pylint` 检测出来的，看上去它将一些微小的问题也检测出来了，而这个是框架生成的代码。

```

No config file found, using default configuration
***** Module growth_studio.urls
C: 19, 0: Invalid constant name "urlpatterns" (invalid-name)
***** Module growth_studio.wsgi
C: 16, 0: Invalid constant name "application" (invalid-name)

```

pylint 的安装方式和 pep8 一样，直接用 `pip install pylint` 命令，再把它加入 `requirements.txt` 中即可。pylint 有一个名为 `pylint_django` 的插件，它可以帮助检测好 Django 编写的代码。

除了 pep8，还有一个更好的选择是：Flake8，它包装了静态检查 PEP 8 编码风格的工具 pep8、静态检查 Python 代码逻辑错误的工具 PyFlakes、静态分析 Python 代码复杂度的工具 Ned Batchelder's McCabe script。

4.2.2 软件包管理

我们在第 1 章提到了包管理的概念，在这里要做的也是类似的事：打包我们的软件包，再发布到包管理中心。通过此我们就可以在产品环境或者测试环境下直接安装该软件包。

在刚开始学会在服务器上部署代码的时候，我都是在本地压缩好开发完的软件包，再通过 `scp` 命令复制到远程的服务器上，最后在服务器上运行部署操作。

后来，我学会了使用 Git 和 GitHub，于是就在本地提交代码，在服务器上拉取最新的代码。

在我实习的项目里，我们使用 GitHub 的标签（Tag）系统来管理和发布软件包。基于 GitHub 标签系统，可以制定一个简单的版本管理流程。我们在第 1 章里提到了基于 Git 的工作流的概念。在发布新的软件版本时，我们会对其标上一个对应的版本号，而 Git 也能支持这种在某一时间点上的版本打标签的功能。

如果我们缺少持续集成服务器，可以在本地打一个 Tag，再将 Tag 提交到远程 Git 服务器上，如 GitHub。我们就可以通过 GitHub 来下载这次发布，并部署到测试或者生产环境中。可以通过下面的命令来创建一个新的标签，即版本 0.0.1：

```
$ git tag v0.0.1
```

再把它提交到远程服务器上：

```
$ git push origin v0.0.1
```

这时就可以从 GitHub 上的 release 页面查看到相应的 release，如图 4-3 所示。



图 4-3 Growth Studio GitHub release 页面

因此，我们可以创建一个相应的任务来执行创建标签任务，代码如下：

```
@task
def tag_version(version):
    "Tag New Version"
    local("git tag %s" % version)
    local("git push origin %s"%version)
```

同时 GitHub 会为我们打的标签创建一个相应的下载地址，如 https://codeload.github.com/phodal/growth_studio/tar.gz/ + 版本号，因此，可以直接下载这个版本的软件包。

```
@task
def fetch_version(version):
    "Fetch Git Version"
    local('wget https://codeload.github.com/phodal/growth_studio/tar.gz/
%s'%version)
```

这样我们就可以完成简单的软件包的发布管理。

4.3 小结

你可能也从不同的渠道了解到一句话：

对同等经验的两个不同的程序员而言，在效率和质量上可能会有 10 倍的差距。

在这 10 倍中，我相信有一半的因素是和所使用的工具有关。在前三章，我们讨论的主要话题都是如何选择好工具来提高开发效率。而本章介绍了一个应用构建系统的组成，以及如何搭建这样的构建系统。我们关注于如何打造好工具来提高开发效率。这也是我们将这一部分称为 **Prepare** 的原因，先缩短好可以缩短的开发效率差距，再一步步赶上。从选择、使用工具，到自己动手做这样一个工具，是一种很不错的体验。

我们在做事的时候,总会习惯性地将一些步骤一步步地执行,对编码来说也是如此。会遵循一定的流程来完成整个编码的过程。我们平时的编码过程如图 5-1 所示。

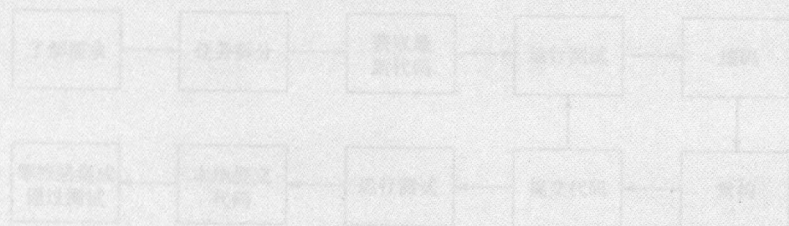


图 5-1 编码流程图

第 2 部分 编码到上线

①了解需求。在这一步要弄清楚我们需要做的事情和验收条件。

②任务拆分。任务应该怎么拆,一般来说,如果是绝对清晰的话,本章

③获取最新代码。对使用 Git 来管理项目的团队来说,在一个任务刚开始的时候应该保证本地的代码是最新的。

④运行测试。测试首先是一个很不错的实践,可以防止代码的bug能避免,并且测试尽可能少,当然也会有测试。

⑤编码。在编写代码的时候,请牢记一些基本原则,本章

⑥发布。完成了上面所有的步骤之后,我们就可以发布代码了,本章

⑦提交代码。这里提交代码只是本地的,因此,提交本地代码提交代码。

⑧运行测试。完成任务后,就可以部署提交代码了,这时需要在本地运行测试,以保证不破坏代码。

⑨本地提交代码。

部署 CI (持续集成) 测试通过。如果这时 CI 是挂掉的话,就需要修复 CI。这时其他的人就没有理由提交代码。如果他们提交的代码也是有问题,这只会使情况变得更加复杂。

第 5 章

编码

在本章，我们将开发实现几个业务功能，并介绍 **Tasking** 和分析的重要性，以及在完成编码前后，编写测试有什么区别、测试和重构将如何改善代码质量。

我们在做事的时候，总会习惯性地按一些步骤一步步往下执行。对编码来说也是如此，会遵循一定的流程来完成整个编码的过程。我们平时的编码过程如图 5-1 所示。

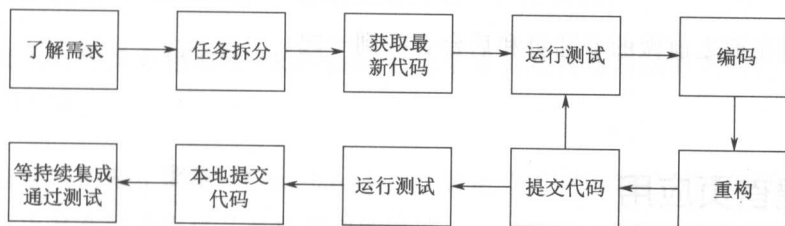


图 5-1 编码过程

这些步骤几乎是一步步往下执行的（其中的测试有可能是在编码后）：

- ①了解需求。在这一步要详细了解我们需要做的事情和验收条件。
- ②任务拆分。简单总结需要怎么做。一般来说，如果是结对编程的话，还会记录过程。
- ③获取最新代码。对使用 Git 来管理项目的团队来说，在一个任务刚开始的时候应该保证本地的代码是最新的。
- ④运行测试。测试优先是一个很不错的实践，可以保证书写的代码的健壮，并且函数尽可能小，当然也会有测试。
- ⑤编码。在编写代码的过程中，请记住一步一步提交代码，一次一个任务。
- ⑥重构。实现了上面两步之后，还需要重构代码，使代码更容易阅读、更易懂等。
- ⑦提交代码。这里提交代码只是本地的，因此，提倡在本地多次提交代码。
- ⑧运行测试。完成任务后，就可以准备提交代码了。这时需要在本地运行测试，以保证不破坏功能。
- ⑨本地提交代码。
- ⑩等 CI（持续集成）测试通过。如果这时 CI 是挂的话，就需要再修 CI。这时其他的人就没有理由提交代码，如果他们的代码也是有问题的，这只会使情况变得更加复杂。

在不同的团队里会有不同的流程和规范。尽管有时候团队的流程并不是那么规范，但在其所处的环境下有可能是最合理的。

下面我们先把之前做的着陆页和后台合并到一起。

5.1 创建首页应用

在决定要做首页页面之前，应该规划好制作步骤。如果你看到目录，可能就知道一个大概情况。

- 创建首页应用。
- 合并着陆页。
- 编写单元测试。
- 使用 Selenium 进行功能测试。
- 编写自动测试的 Fabric 任务。

我们强调 Tasking 的重要性，不仅是因为可以细分好任务，而且当我们在实践每一小步的时候，都知道当前正在做什么。同时对应于每一步，可以以此为依据来提交代码，我们所要写的提交信息就是这一步内容。每次在本地提交代码的时候，步伐小就意味着：一旦出错，就可以回滚到上次的提交，其重要性类似于在玩游戏时存档的重要性。

我们将这种一点点前进的步骤称为小步前进，每一步都有明确的任务，每个任务有相应的衡量指标。如这里的创建首页应用，可以对其进行衡量。不过，这种小的目标与本节的任务“创建首页”的目标稍有不同。对这种更大的目标，我们会遵循 SMART 原则（S=Specific、M=Measurable、A=Attainable、R=Relevant、T=Time-based）来进行目标管理，以这里创建首页为例。

- 具体的（Specific）：需要做的就是创建一个首页应用，并合并原来的着陆页。

- 可以衡量的 (Measurable): 当程序运行后, 我们可以在浏览器上访问首页。
- 可以达到的 (Attainable): 这毫无疑问是可以做到的。
- 和其他目标具有相关性 (Relevant): 这个首页是其他页面的入口, 介绍了项目相关的内容。
- 具有明确的截止期限 (Time-based): 尽管这里没有明确的时间期限, 但这个期限实际上是在阅读完这几节所需要的时间。

这里只对 SMART 原则做简单介绍, 有兴趣的读者可以深入这方面的了解。

对小的任务来说, 在时间上有一些不确定性。但对前四者来说, 它也是应该需要达到的。当大的任务被拆分为小任务时, 每完成一小步就会发现在接近目标。当遇到问题时, 也很容易明白现在遇到的问题是什么。

下面让我们再回到创建首页的这件事情上。

5.1.1 生成首页应用

在 Django 里每个小的模块都被称为 App, 我们将其称为应用。这些应用都有一些基础文件, 并且都可以通过 Django 命令行工具创建出来。因此, 可以直接使用 `startapp` 来创建一个 `homepage` 的应用:

```
$ django-admin startapp homepage
```

它将生成如下所救援的 `homepage` 文件夹及以及相关的代码文件:

```
.
├── __init__.py
├── admin.py
├── apps.py
├── migrations
└── __init__.py
```

```
├─ models.py
├─ tests.py
└─ views.py
```

1 directory, 6 files

在其他 MVC 框架里,如 Spring MVC,所有的控制器一般都会放在 `controllers` 里,model 会放在对应的 `models` 文件夹里。但是在 Django 里,每个 App 只含有自己所需要的 MVC。在自己的 `views.py` 里放置对应的业务逻辑,在自己的 `models.py` 文件里放置自己所需要的模型,在公有的 `templates` 文件里放置业务的模板。

由于 `homepage` 只用于展示首页,因此可以只保留 `views.py`、`tests.py`,以及用于标注这是一个 Python 模块的 `__init__.py` 三个文件。

建议读者在删除之前,先提交代码,提交信息可以类似于 `init homepage app`。

接着,需要简单的配置,以便让系统加载这个应用。我们所需做的就是 `settings.py` 文件的 `INSTALLED_APPS` 字段中添加 `homepage`,代码如下:

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'homepage'
]
```

至此,我们已经添加了首页应用到程序里,但并没有看到任何现象。那么,就让我们继续创建一个用 HTML 编写的“hello, world”。

1. 创建一个 hello, world

在第 2 章中,我们提到在访问一个网站时需要先经过 HTTP 服务器,再由 HTTP 服

务器返回对应的 HTML 内容。这时，我们需要对 HTTP 服务器进行配置，才会有相应的 URL 返回对应的内容。

对 Web 应用也是如此，在应用的内部会有对应的 URL 调度器（Dispatcher）来处理分发 URL。在不同的 Web 框架里，这个 URL Dispatcher 有不同的名称，如 Spring 框架里叫 Dispatcher Servlet，但它们都会具有相似的作用。如在 Django 里，只需要配置 `urls.py`，就可以将访问 homepage 的 URL 交给 homepage 应用来处理，可以将访问 `blog/blog-1.html` 及 `blog/blog/blog-2.html` 统一交由 blog 应用来处理。而应用则会将请求交由应用中对应的 View 来处理，View 再根据需求来填充数据返回给浏览器，其过程如图 5-2 所示。

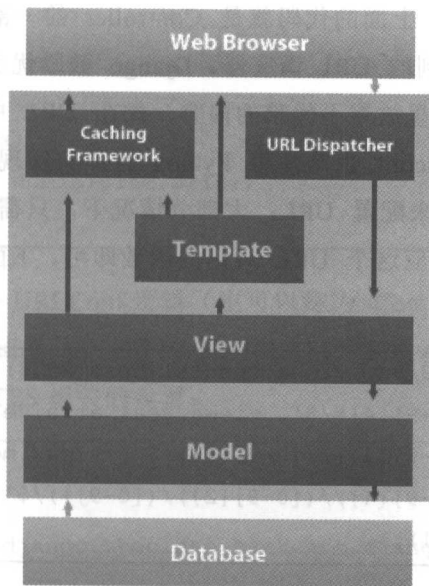


图 5-2 Django URL Dispatcher

在这里，这个步骤就是：

①在 URL dispatcher 中指向对应的 View 中的函数。

②由 View 中对应的函数来指定返回的内容，即 HTML 文件。

除此之外，我们还需要定义项目中模板文件的路径。

由于需要先有 View 才能配置好 URL，因此，先创建这个 View。由于返回的内容只是一个静态的 HTML，因此只需要调用 Django 中的 `render_to_response` 方法来直接返回模板即可。代码如下：

```
from django.shortcuts import render_to_response

def index(request):
    return render_to_response('index.html')
```

在其他 MVC 框架里，上面的代码就是 Controller 的一部分，还需要的配置就是对 URL 的处理，即在文中说到的 URL 调度器。Django 鼓励优美的 URL 设计，并且不会像一些 Web 框架在 URL 里放置不优雅的内容，如在 URL 中的 `.php` 或者 `.asp`。更有意思的是，Django 的 `URLconf` 也是使用 Python 代码来实现的。Django 使用一个名为 `django.conf.urls.url()` 的方法来配置 URL，大部分情况下，只需记住这个方法是传入 URL 对应的正则表达式，以及对应这个 URL 规则的函数即可。下面是 Django 官方文档里给出的几个 URL 示例：

```
url(r'^articles/2003/$', views.special_case_2003),
url(r'^articles/([0-9]{4})/$', views.year_archive),
url(r'^articles/([0-9]{4})/([0-9]{2})/$', views.month_archive),
url(r'^articles/([0-9]{4})/([0-9]{2})/([0-9]+)/$',
views.article_detail),
```

这是一个处理不同年、月、日以及特定日期的 URL 请求。上面代码中的 `r` 表示是一个只读字符串，即不需要转义；代码中的 `$` 则是字符串结束匹配符，表明这个正则表达式到这里结束。

因此，在第一个例子中的表达式 `r'^articles/2003/$'` 将会匹配特定的 URL，即 `/articles/2003/`，并交给 `views` 模块（即 `views.py` 文件）中的 `special_case_2003` 方法来处理。因为每个 URL 都会以斜杠 “/” 开头，因此，在这个框架里，每个正则表达式的斜杠 “/” 可以交由框架来处理。

由于在 `URLConf` 中会按由上往下的顺序执行，因此，上述 URL `/articles/2003/` 将会交由第一个表达式来处理。而当我们访问 `/articles/2004/` 时，才会交由第二个表达式 `^articles/([0-9]{4})/$` 来匹配，并交由 `year_archive` 方法来处理。以此类推，可以得到所需要的 URL 规则。

这里需要的 URL 有些特殊——需要直接映射到首页，这时只需要这样的表达式 `^$`，故只需要引入 `index` 函数，并添加上述正则表达式即可。最后的 `urls.py` 如下：

```
from django.conf.urls import url
from django.contrib import admin
from homepage.views import index

urlpatterns = [
    url(r'^$', index),
    url(r'^admin/', admin.site.urls),
]
```

在完成对应的 View + `URLConf` 逻辑（也可以称为 `Controller`）后，就可以对模板进行处理了。我们需要将 `settings.py` 文件的 `TEMPLATES` 字段设置为 `DIRS` 添加值，即模板的路径。下面是 `TEMPLATES` 相关的设置：

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.debug',
                'django.template.context_processors.request',
                'django.contrib.auth.context_processors.auth',
                'django.contrib.messages.context_processors.messages',
            ],
        },
    },
]
```

```
    },  
    },  
  ]  
}
```

一般来说，设置这个路径为 `templates`，因此其配置就是：

```
DIRS': ['templates/'],
```

接着可以创建对应的文件夹，并在文件夹里创建一个 HTML 文件 `index.html`。添加代码如下：

```
<!DOCTYPE html>  
<html lang="en">  
<head>  
  <meta charset="UTF-8">  
  <title>hello, world</title>  
</head>  
<body>  
  <h1>hello, world</h1>  
</body>  
</html>
```

现在可以运行这个 Demo 了，我们将会看到浏览器上这个“hello, world”。它的运行逻辑和我们预期的一样，当我们访问 `http://127.0.0.1:8000` 之后，Django 框架将会读取不同 `urls.py` 文件中的 `urlpatterns` 配置，找到相应的 URL 表达式后，再交由相应的函数来处理。

2. 合并着陆页

在使用 Django 开发 Web 应用时，通常会在项目的目录下创建一个 `static` 文件夹，用于放置静态文件。我们会在这个文件夹下放置 `js`、`css`、字体文件等静态资源。因此，我们可以将在第 2 章写好的 `index.html` 以及那些静态文件复制到项目里，变成如下所示的目录结构：

```

├── static
│   ├── css
│   │   ├── bootstrap-theme.min.css
│   │   ├── bootstrap-theme.min.css.map
│   │   ├── bootstrap.min.css
│   │   ├── bootstrap.min.css.map
│   │   └── carousel.css
│   ├── fonts
│   │   ├── glyphicons-halflings-regular.eot
│   │   ├── glyphicons-halflings-regular.svg
│   │   ├── glyphicons-halflings-regular.ttf
│   │   ├── glyphicons-halflings-regular.woff
│   │   └── glyphicons-halflings-regular.woff2
│   ├── js
│   │   └── vendor
│   │       ├── bootstrap.min.js
│   │       ├── holder.min.js
│   │       └── jquery-3.1.1.min.js
├── templates
└── index.html

```

当我们把 HTML 从着陆页复制过来时，还需要向配置文件 `settings.py` 中添加相应的配置指定好静态文件加载的目录。在 Django 里使用 `STATICFILES_DIRS` 来设置静态文件的目录，代码如下：

```

STATICFILES_DIRS = (
    os.path.join(BASE_DIR, 'static/')
)

```

这里的 `BASE_DIR` 是项目的相对位置，`os.path.join` 方法会合并这两个路径，即变成当前项目下的 `static` 文件夹。这样静态文件就会被成功识别，所需要做的就是运行这个服务即可。

让我们做一次代码提交，完成着陆页的代码设计，接着做一些轻松的编码工作，编写测试。

5.1.2 编写第一个测试

Django 自带了测试组件，它可以让我们很方便地对代码进行测试，我们来看一个测试的例子。编辑 `homepage` 模块下的 `tests.py` 文件，并添加如下内容：

```
from django.core.urlresolvers import resolve
from django.http import HttpRequest
from django.test import TestCase

from homepage.views import index

class HomePageTest(TestCase):
    def test_root_url_resolves_to_home_page_view(self):
        found = resolve('/')
        self.assertEqual(found.func, index)
```

我们要写的第一个测试比较简单，即 `test_root_url_resolves_to_home_page_view`，当我们访问根目录的时候，应该被定向到首页的 `view` 方法。代码中使用了 `resolve` 来获取 '/' 这个 URL 对应的 `URLConf`，并使用 `assertEqual` 方法来断言这个 URL 对应的函数是 `index` 方法。当这个断言成功时，测试将会通过。

现在，只需要运行 `python manage.py test` 就可以运行测试：

```
Creating test database for alias 'default'...
```

```
.
```

```
-----
Ran 1 test in 0.004s
```

```
OK
```

```
Destroying test database for alias 'default'...
```

断言失败时，会告知哪个测试是有问题的，结果如下：

```

Creating test database for alias 'default'...
E
=====
ERROR: test_root_url_resolves_to_home_page_view (homepage.tests.HomePageTest)
-----
Traceback (most recent call last):
  File "/Users/fdhuang/write/growth_studio/homepage/tests.py", line 10, in
test_root_url_resolves_to_home_page_view
    found = resolve('/home')
  File
"/Users/fdhuang/write/py35env/lib/python3.5/site-packages/django/urls/ba
se.py", line 27, in resolve
    return get_resolver(urlconf).resolve(path)
  File
"/Users/fdhuang/write/py35env/lib/python3.5/site-packages/django/urls/re
solvers.py", line 300, in resolve
    raise Resolver404({'tried': tried, 'path': new_path})
django.urls.exceptions.Resolver404: {'tried': [[<RegexURLPattern home ^$>],
[<RegexURLResolver <module 'django.contrib.flatpages.urls' from '/Users/
fdhuang/write/py35env/lib/python3.5/site-packages/django/contrib/flatpages
/urls.py'> (None:None) ^pages/>], [<RegexURLResolver <RegexURLPattern list>
(admin:admin) ^admin/>]], 'path': 'home'}

-----
Ran 1 test in 0.006s

FAILED (errors=1)
Destroying test database for alias 'default'...

```

我们就能根据错误信息 `ERROR: test_root_url_resolves_to_home_page_view(homepage.tests.HomePageTest)` 找到对应的方法，并根据下面的 404 异常来知道原因。

除此之外，我们还能发现在最开始测试的时候，Django 的测试组件会创建一个测试用

的数据库，并在测试结束的时候将这个数据库删除。这个步骤可以帮助我们在测试的时候隔离运行测试时用的数据和在本地用的测试数据。

我们也可以将上述脚本简单写成一个任务，以便以后在持续集成服务器上直接执行。

```
@task
def test():
    """ Run Test """
    local("./manage.py test")
```

我们还可以用它来判断页面返回的内容是不是我们想要的。上面的测试过于简单，只能验证过程是否正确，无法验证返回结果。下面让我们看一个难返回结果的例子：

```
def test_home_page_returns_correct_html(self):
    request = HttpRequest()
    response = index(request)
    self.assertIn(b'<title>Growth Studio - Enjoy Create & Share</title>',
                  response.content)
```

这段代码用于验证首页的标题是否为 HTML 中写的内容，即 Growth Studio - Enjoy Create & Share。而这里断言方式则是判断页面返回的 HTML 包含（即 `assertIn` 方法）我们的标题。

5.1.3 使用 Selenium 进行功能测试

与上面的单元测试相比，使用 Selenium 编写功能测试可能更有挑战性，并且更具有测试的说服力。Selenium 是一个 Web 应用程序测试框架，它可以让浏览器自动化地执行任务。借助于 Selenium，我们可以通过代码打开某个特定的网站，在网站的表单输入框里输入内容，然后单击表单下面的提交，最后查看页面是否会有相应的结果产生。

在这里，我们要做的就是通过 Selenium 来打开首页，并判断首页的标题是不是我们想要的内容。

为了使用 Selenium，我们需要安装不同语言对应的 Selenium——实际上是 Selenium 为不同语言封装好的 API。我们只需要通过这些 API 来操作浏览器，其安装方法依然很简单：

```
$ pip install selenium
```

记得将这个包添加到 requirements.txt 文件中。同时，对应不同的浏览器，我们还需要安装不同的 webdriver，如 Chrome 浏览器里，我们需要下载 chromedriver，Firefox 浏览器需要 geckodriver。

这取决于你将使用哪个浏览器来运行测试，需要注意的是，不同版本的 Driver 支持不同版本的浏览器。如果在初始化的时候遇到一些问题，则有可能就是版本不对应导致的。如在笔者写作时：ChromeDriver 2.25 对应支持的是 Chrome v53-55，而 ChromeDriver 2.24 对 Chrome 53 以上的版本可能存在问题。

可以从本书源代码的 GitHub 页面获取下载地址，或者获取存档在 GitHub 上的备份。

下面看一个使用 selenium 写的功能测试，代码如下：

```
from django.test import StaticLiveServerTestCase
from selenium import webdriver
```

```
class HomepageTestCase(StaticLiveServerTestCase):
```

```
    def setUp(self):
```

```
        self.selenium = webdriver.Chrome()
```

```
        self.selenium.maximize_window()
```

```
        super(HomepageTestCase, self).setUp()
```

```
    def tearDown(self):
```

```
        self.selenium.quit()
```

```
        super(HomepageTestCase, self).tearDown()
```

```
    def test_can_visit_homepage(self):
```

```
self.selenium.get(
    '%s%s' % (self.live_server_url, "/")
)

self.assertIn("Growth Studio - Enjoy Create & Share", self.selenium.
title)
```

我们的测试继承自 `LiveServerTestCase` 类，该类将在运行的时候创建一个临时数据库，并运行一个测试服务器。在 `LiveServerTestCase` 类里，它将按顺序执行一些步骤，然后执行测试。

在上面的代码里主要有三个方法：`setUp()`、`tearDown()`和 `test_can_visit_homepage()`。在这三个方法中起主要作用的是 `test_can_visit_homepage()`方法。而 `setUp()`和 `tearDown()`是特殊的方法，分别在测试方法开始之前运行和之后运行。同时，在这里也用这两个方法来打开和关闭浏览器。

在我们的测试方法 `test_can_visit_homepage()` 里，主要有以下两个步骤。

①访问首页。

②验证首页的标题是 “Growth Studio - Enjoy Create & Share”。

大部分测试代码也是以相似的流程来运行的。有一点需要注意的是：一般来说，函数名就表示测试所要做的事情，如这里测试的就是可以访问首页。

如果在运行过程中遇到 “`socket.gaierror: [Errno 8] nodename nor servname provided, or not known`” 的错误，请在你的 `hosts` 文件中添加一行 `127.0.0.1 localhost`。

如上所示的测试过程称为 “四阶段测试”，即这个过程分为如下四个阶段。

①**Setup**。在这个阶段主要是做一些准备工作，如数据准备和初始化等，在上面的 `setup` 阶段就是用 `selenium` 启动一个 `Firefox` 浏览器，然后把窗口最大化。

②**Execute**。在执行阶段就是做好验证结果前的工作，如在测试注册的时候，这里是填写数据，并单击提交按钮。上面的代码只是打开了首页。

③**Verify**。在验证阶段所要做的就是验证返回的结果是否和预期的一致。这里还是使用和单元测试一样的 `assert` 来做断言,通过判断这个页面的标题是“Welcome to my blog”,来说明我们现在位于首页。

④**Tear Down**。就是一些收尾工作,比如关闭浏览器、清除测试数据等。

事实上,这些步骤也是在遵循小步前进的原则,每一小步只做一件事,每件事都有其意义。

需要注意以下几点。

①从运行测试速度上看,三种测试的运行速度呈倒金字塔结构。即,单元测试跑得越快,开发速度也越快;随后是服务测试,最后是 UI 测试。

②即使现在的 UI 测试跑得非常快,但随着时间的推移,UI 测试会越来越多。这也意味着测试跑得越来越久,那么人们就开始不想测试了。在我们之前的项目里,运行完所有的测试大概接近一个小时,我们开始在会上争论这些测试的必要性,也在想方设法减少这些测试。

③如果一个测试可以在最底层写,那么就不要在它的上一层写了,因为它的运行速度更快。

你可能注意到了一点,我们使用 Django 自带的测试组件编写的单元测试与 Selenium 写出来的测试差不多——断言页面返回的 HTML 的标题都是 Growth Studio-Enjoy Create & Share。在这里只是因为测试页面,只有首页比较单一。在下面的内容里将会看到:

①Django 自带的测试组件所编写的测试限制于这个页面的内容里,并且限制于静态的内容。当对页面进行操作的时候,我们难以追求内容的变化,特别是使用 JavaScript 来操纵页面元素时。

②Selenium 框架里自带了对 HTML 进行解析的组件,这就是为什么我们看到的 Django 自带测试组件编写的测试里还有 HTML 标签。同时,我们还可以使用它来测试页面的点击和跳转,这些都是模拟真实的人为操作。当有大量的 JavaScript 代码来操作页面时,就能体会到它的强大之处。

5.1.4 如何编写测试

多数时候，写测试相比于写代码来说相对较简单——并不需要考虑复杂的逻辑，只需要验证输出结果是正确的，并按照代码逻辑，对代码的行为进行覆盖。需要注意的是，在不同的团队、工作流里，测试可能会有不同的工作流程：

- 开发人员写单元测试、集成测试等。
- 测试团队通过界面来做黑盒测试。
- 测试人员手动测试来测试功能。

在允许的情况下，测试应该由开发人员来编写，并由底层开始写测试。

1. 为什么需要测试

我们很容易从开发者那里收集到足够的理由来说明他们为什么不写测试，如：

- 编写测试需要花费时间，这将影响项目的交付。
- 项目没有对单元测试进行要求。
- 单元测试没有业务价值。

要找到一个合理的理由是一件容易的事。对我而言，编写测试具备下面一些好处：

- 保证现有代码的功能都是正常的。当我修改代码时，可能就破坏了现有的功能，但我可能没有意识到这个问题，在测试的时候就会告诉我们这里的测试挂了，我才知道我的修改已经影响到这个地方的代码了。
- 帮助找到代码中的 bug。在编写单元测试时，我们测试的用例（即针对某种情况下的测试代码）都是对照某一种情况下而编写的。如编写一个将数字转换为汉语的程序时，就需要针对万、亿等不同级别的转换做相应的测试，对不同的转换都要有对应的测试来覆盖。因此，我们很容易在编写代码的时候发现缺少了某部分的用例。

- 编写出长度短小的代码。复杂的代码不容易写出测试，拥有相当高测试覆盖率的代码也不会过于复杂——对代码中的嵌套语句都需要有相应的测试，复杂的嵌套语句在测试时就会遇到一些障碍，这时理想的方式就是将其提炼为函数。
- 为重构代码打下基础。如果你对代码重构有一定的了解，就会有一些体会。当我们进行代码重构时，最害怕发生的事情就是修改的代码会破坏已有的功能。测试在这时可以保证现有的代码功能都是正常的，你就可以放心大胆地重构了。

在开始编写测试之前，我们需要了解测试金字塔，它表明了不同层级的测试之间的关系。

2. 测试金字塔

测试金字塔是由 Mike Cohn 提出的，主要观点是：底层单元测试应多于依赖 GUI 的高层端到端测试，其结构如图 5-3 所示。



图 5-3 测试金字塔结构

从结构上看，图 5-3 的金字塔可以分成三部分：单元测试、服务测试、UI 测试。单元测试应该是最多的，也是最底层的。其次是服务测试，最后是 UI 测试。大量的单元测试可以保证我们的基础函数是正常且正确工作的。而服务测试则是一门很有学问的测试，不仅只测试我们自己提供的服务，也会测试依赖第三方提供的服务。在测试第三方提供的服务时，就会变成一件有意思的事情。除此之外，还有对功能和 UI 的测试，写这些测试可以减轻测试人员的工作量。

(1) 单元测试

单元测试是针对程序模块（软件设计的最小单位）进行正确性检验的测试工作。它是

应用的最小可测试部件。举个例子，下面是 Python 代码实现的一个加法函数。

```
def add(a, b):  
    return a + b
```

这是一个很简单的功能，对应有一个断言来确保返回的结果是我们所需要的：

```
def should_return_correct_result(self):  
    result = self.calc.add(1, 2)  
    self.assertEqual(3, result)
```

这个测试看上去很简单，但大量基本的单元测试可以保证调用的函数都是可以正常工作的。这也相当于在建设金字塔时用的石块——如果石块都是经过检验的，那么就不怕金字塔因为石块的损坏而坍塌。

当单元测试达到一定的覆盖率时，我们的代码就会变得更健壮。因为我们需要保证代码都是可测的，也意味着代码间的耦合度会降低。我们需要考虑代码的长度，越长的代码，其测试的时间会变得越困难。这也就是为什么 TDD 会促使我们写出短的代码。如果我们的代码都是经过测试的，单元测试可以帮助我们在未来重构代码。

在很多没有文档或者文档不完整的开源项目中，了解这个项目某个函数的用法就是查看其测试用例。测试用例（Test Case）是为某个特殊目标而编制的一组测试输入、执行条件以及预期结果，以便测试某个程序路径或核实是否满足某个特定需求。这些测试用例可以让我们直观地理解程序的 API。

（2）服务测试

顾名思义，服务测试就是对服务进行测试，而服务可以有不同的类型，不同层次的测试。如第三方的 API 服务、程序提供的服务，虽然它们应该在这个层级上进行测试，但其测试会稍有不同。

对于第三方提供的 API 服务或者其他类似的服务，在这一个层级的测试都不会真实地去测试它们能不能工作——这些依赖性的服务只会在功能测试上进行测试。在这里的测试只会保证功能代码是可以正常工作的，所以我们会使用一些虚假的 API 测试数据来进行测

试。这一类提供 API 的 Mock Server（见图 5-4）可以模拟被测系统外部依赖模块行为的通用服务。我们只要保证功能代码是正常工作的，那么依赖它的服务也会是正常工作的。

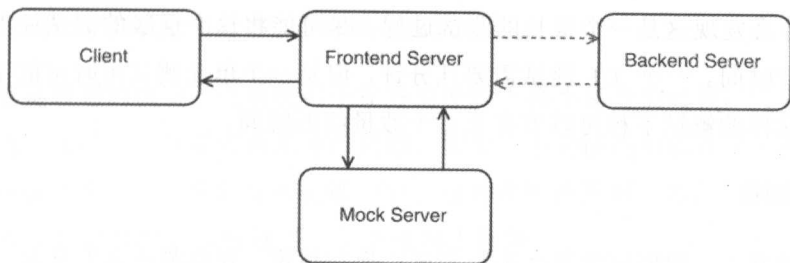


图 5-4 Mock Server

对我们提供的服务来说，这一类服务不一定是 API 的服务，还有可能是多个函数组成的功能性服务。当我们在测试这些服务的时候，实际上是在测试这个函数结合在一起是不是正常的。

一个服务可能依赖于多个函数，因而我们会发现服务测试的数量是少于单元测试的。

（3）UI 测试

在 Web 开发领域，UI 测试又可以称为功能测试、验收测试、端对端测试。这时我们不关心系统内部的组成，只关心它在功能上是否正确，是否能正确响应对应的行为。

在传统的软件开发中，UI 测试多数是由人手动来完成的。而在稍后的章节里，你会看到这些工作是可以由机器来完成的。当然，前提是我们要编写这些自动化测试的代码。需要注意的是，UI 测试并不能完全替代手工工作，一些测试还是应该由人来进行测试，如对 UI 的布局，现阶段机器还没有审美意识。

自动化 UI 测试是一个缓慢的过程，在这个过程里需要做以下事情。

①运行网站——这可能需要几分钟。

②添加一些 Mock 的数据，以使网站看上去正常——这也需要几分钟到几十分钟的时间。

③开始运行测试——在一些依赖于网络的测试中，运行完一个测试可能需要几分钟。尽管可以并行运行测试，但是一个测试几分钟算到最后就会累积成很长的时间。

所以，你会发现这是一个很长的测试过程。尽可能将这个层级的测试往下层级移，就会尽可能节省时间。一个 UI 测试需要几分钟，但是一个单元测试用时可能不到 1 秒。这就意味着，这样的测试下移可以节省上百个数量级的时间。

3. 如何测试

现在问题来了，我们应该怎么去写测试？换句话说，我要测什么？这是一个很难的问题，这足够可以写一本书来说明这个问题。这个问题也需要依赖于不同的实践，不同的时候可能对问题的看法不同。

编写测试的过程大致可以分成下面几个步骤：

①了解测试目的（Why）。即需要测什么，为什么编写测试。

②要测哪些内容（What）。即测试点，从功能点出发来寻找需要测试的点，在不同的条件下这个测试点是不一样的。

③要如何进行测试（How）。使用什么方法进行测试？

（1）测试目的

我们在上面提到过的测试金字塔也表明了在每个层级要测试的目的是不一样的。

在单元测试这一层级，因为所测试的是每一个函数，这些函数无法构成完整的功能。这时就只是用于简简单单的测试函数本身的功能，没有太多的业务需求。

而对于服务层级，所要测试的就是一个完整的功能。对以 API 为主的项目来说，实际上就是在测返回结果是否正确。

最后是 UI 层级，我们需要测试的就是一个完整的功能。用户操作的时候应该是怎样的，就应该模仿用户的行为来测试。这是一个完整的业务需求，也可以称为验证测试。

(2) 测试点

在了解完要测试的目的之后，要测试的点也变得很清晰。即在单元测试中测试函数的功能，在服务测试中测试服务，在 UI 测试中测试业务。

而这些都是理想的情况，当系统由于业务的原因不得不耦合的时候，究竟是单元测试还是功能测试，这是一个特别值得思考的问题。如果一个功能既可以在单元测试里测，又可以在服务测试里测，那么我们要测试哪一个？或者说应该把两个都测一遍？而如果是花费时间更长的 UI 测试呢？这样做是不是会变得不划算。

(3) 如何测试

对不同功能的函数，我们会使用不同的测试方法来进行测试。常见的有：

- 状态测试。关注于代码返回的结果，预期代码会返回我们所需要的结果。
- 行为测试。关注于代码执行的过程，预期代码在执行过程中会调用指定的函数，即执行指定的行为。

再回到上面那个单元测试的例子，我们只需要判断调用这个方法时，它能返回我们预期的结果。如输入的是 1 和 2，那么预期它应该返回一个 3，代码如下：

```
def should_return_correct_result(self):  
    result = self.calc.add(1, 2)  
    self.assertEqual(3, result)
```

这是一类比较容易实现的测试，在实现过程中，我们只需要想办法得到想要的结果即可。对行为测试来说，则会稍微麻烦一些。当我们打开浏览器输入某个网站时，只会看到页面上的返回结果。虽然我们看不到这个过程 API 请求，但是可以预期请求 API 的方法被调用了。这时可以仿造这个方法，并预期这个方法被调用。

5.2 创建博客应用

尽管前面做了很多工作，但它仍然是没有业务价值的工作。这种业务与技术的冲突在工作中会特别明显，但它们是相当有益于开发出高质量代码的步骤。下一步就会创建一些有实质性的工作。

5.2.1 创建应用与博客管理

在我们不了解 Django 的时候，要对这样一个任务进行 Tasking（任务拆分）有点困难。不过，我们还是可以简单地看看应该如何做。

- 生成 App。对大部分主流的 Web 框架来说，它们都可以手动生成一些脚手架，如 Ruby 语言中的 Ruby On Rails、Node.js 中的 Express 等。
- 创建对应的 Model，即其在数据库中存储的模型与我们在代码中要使用的模型。
- 创建程序对应的 View，用于处理数据。
- 创建程序的 Template，用于显示数据。
- 编写测试来保证功能。

对其他应用来说也是差不多的：

- 生成基本的脚手架。在这个脚手架里应该包含基本的模型、Controller 的最小逻辑及 URL 处理、基本的模板，我们在这些代码的基础上可以很容易完成对任务的开发。
- 设计和创建数据模型。当我们的应用需要存储数据时，就应考虑数据存储的形式，以及其对应的数据库结构。同时，可以开始创建表，并编写 ORM 层代码。
- 编写业务逻辑。在这一步需要创建对应的 Controller，同时考虑给前台的数据内容，并对不需要的数据进行过滤。

- 完成页面显示及美化。需要先编写 HTML 在页面上显示内容，并编写 CSS 完善 UI、编写 JavaScript 来处理用户交互。

不同的人有不同的风格，我偏爱先完成基本的轮廓，再一步步完善细节。这样做有一个优点是，每一步都可收到足够的反馈。这些反馈可以表明我们所做的是对的，并不需要一一确认其他地方出了错。

1. 生成脚手架

现在我们可以开始创建 App，同样使用 `startapp` 来生成代码：

```
python manage.py startapp blog
```

随后会在 `blog` 目录下生成下面的文件内容：

```
.
├── __init__.py
├── admin.py
├── apps.py
├── migrations
├──   └── __init__.py
├── models.py
├── tests.py
└── views.py
```

接着，将 App 添加到 `settings.py` 中的 `INSTALLED_APPS` 字段里：

```
INSTALLED_APPS = [
    ...
    'homepage',
    'blog',
]
```

这个步骤和创建 `homepage` 时类似，只是这次我们将用到大部分文件，还将使用后台管理界面来创建数据。

2. 设计和创建 Model

下面，需要创建博客的 Model。对一篇基本的博客来说，它会包含下面的内容。

- 标题：对应于 `title`，用于显示在页面上的标题。
- 作者：对应于 `author`，可以直接关联已经创建的用户。
- 链接：对应于 `slug`，显示在浏览器地址栏上的链接，一般是出于 SEO 以及可读性的需求而创建的。
- 内容：对应于 `body`，即博客的内容。
- 发布日期：对应于 `posted`，用于显示博客的发表时间。

按照上面的内容来创建 `blog` 的 Model：

```
# coding=utf-8
from django.contrib.auth.models import User
from django.db import models
from django.db.models import permalink
from django.utils.translation import ugettext_lazy as _

class Blog(models.Model):
    class Meta:
        verbose_name = _('博客')
        verbose_name_plural = _('博客')

    title = models.CharField(max_length=30, unique=True, verbose_name=_('标题'), help_text='博客的标题')
    author = models.ForeignKey(User, verbose_name=_('作者'))
    slug = models.SlugField(max_length=50, unique=True, verbose_name=_('URL'))
    body = models.TextField(verbose_name=_('正文'))
    posted = models.DateField(db_index=True, auto_now_add=True)
```

```
def __str__(self):
    return '%s' % (self.title)

@permalink
def get_absolute_url(self):
    return 'blog_view', None, {'slug': self.slug}
```

在上面的代码里，一共创建了五个不同的 Field，我们以 title 字段为例来说明如何创建一个字段。Django 的 Model 字段依据需要可以有不同的配置，如 verbose_name 参数是用于在后台显示时的字段名，而 help_text 参数顾名思义则是用于显示帮助文本，在后台的显示效果如图 5-5 所示。

The screenshot shows the Django management interface for adding a blog post. The title field has a help text '博客的标题' (Blog title). The form includes fields for title, author, URL, and content.

图 5-5 Django 后台标题及帮助文本显示

同时使用 max_length 来限定 title 的值最多只能为 30 个字符，unique 表明了这个字段的每个值都是唯一的，不能出现重复。在最后的 posted 字段里有一个 auto_now_add 参数，它用来表明自己使用当前时间。除了这些常用的用法，还可以使用 ForeignKey 来关联其他字段，在 author 字段里将作者关联到了已经注册的用户中，也可以使用 choices 参数来添加可选的下拉列表。

除此之外，我们还看到在这个 Model 里有两个方法。

- __str__ 方法用于在 Django 的后台界面显示对象，这里返回的是博客的标题，

即当我们在后台的列表页显示时，将优化显示博客的标题。

- `get_absolute_url` 方法就是用于返回博客的链接。之所以使用手动而不是自动生成，是因为自动生成不靠谱。

接着，可以运行数据迁移命令将模型应用到数据库中：

```
$ python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, blog, contenttypes, sessions
Running migrations:
  Applying blog.0001_initial... OK
```

在这个过程中，我们的数据库将创建表名为 `blog_blog` 的数据库，其中的第一个 `blog` 对应的是应用名，第二个是这里的 `model` 名。即一个应用可以有两个模型，生成的表名将以应用名_表名的形式存在。在这个过程中，它还创建了一个数据迁移文件 `0001_initial.py`，该文件保存了当前的数据模型。

读者可以使用一些数据库工具（诸如 SQLite Browser）来查看数据库中是否有相应的表生成。

在其他一些 Web 框架里，为了验证生成的模型是否正确，就需要编写 SQL 或者数据库工具来创建数据。而对 Django 开发者来说，只需要在后台管理界面创建数据即可。

3. 后台管理

Django 提供了一个非常强大的后台管理组件，即 `admin`，它可以让我们在后台管理模型，并且可以创建不同的用户组、角色来管理不同的模型。当我们使用 `startproject` 来创建项目的时候，它将自动配置好 `admin` 组件的相关配置。我们所需做的只是：在 `blog` 应用的 `admin.py` 文件中向后台管理界面注册这个 `Model`，代码如下：

```
from django.contrib import admin
from blog.models import Blog
```

```
admin.site.register(Blog)
```

现在登录后台，就可以看到 Blog 一栏，如图 5-6 所示，表示可以对其进行相关的操作。

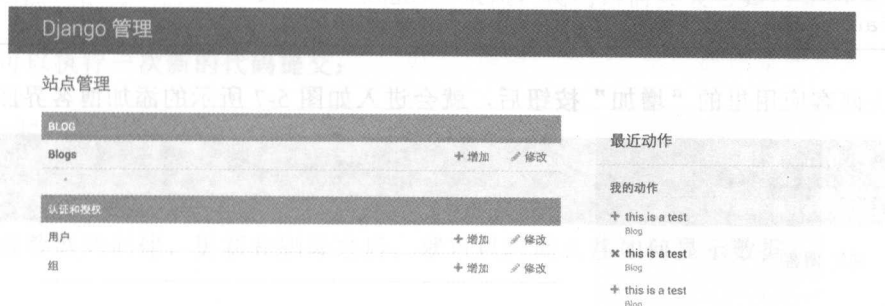


图 5-6 Django 后台界面

我们可以在 Blog 模型里添加一些元信息来修改它在管理界面的显示情况，代码如下：

```
class Blog(models.Model):
    class Meta:
        verbose_name = _('博客')
        verbose_name_plural = _('博客')
```

这里的 `verbose_name` 是用于显示在第二个 Blog 的标题，`verbose_name_plural` 表示其复数形式下显示的名字。

当我们需要显示中文时，可在 `*.py` 文件的第一行添加 `# coding=utf-8` 来指明文件使用 UTF-8 编码以支持中文显示。我们还可以修改浅蓝色的 Blog 为博客应用，只需要修改 `apps.py` 文件为：

```
# coding=utf-8
from django.apps import AppConfig

class BlogConfig(AppConfig):
    name = 'blog'
    verbose_name = "博客应用"
```

这里的 `verbose_name` 同样为显示在页面上的名字，我们还需要在 `__init__.py` 文件中添加指定应用默认配置的类：

```
default_app_config = 'blog.apps.BlogConfig'
```

单击博客应用里的“增加”按钮后，就会进入如图 5-7 所示的添加博客界面。

The screenshot shows the 'Django 管理' (Django Management) interface. The breadcrumb trail is '首页 > 博客应用 > 博客 > 增加 博客' (Home > Blog App > Blog > Add Blog). The main heading is '增加 博客' (Add Blog). There are four input fields: '标题:' (Title) with the value 'this is a test', '作者:' (Author) with a dropdown menu showing 'phodal', 'URL:' with the value 'this-is-a-test', and '正文:' (Body) with the value 'hello, world'.

图 5-7 Django 添加博客

单击“保存”按钮后，即可跳转到如图 5-8 所示的页面。

The screenshot shows the 'Django 管理' (Django Management) interface after saving. The breadcrumb trail is '首页 > 博客应用 > 博客' (Home > Blog App > Blog). A success message is displayed: '✓ 博客 "this is a test" 添加成功。' (Blog "this is a test" added successfully). The main heading is '选择 博客 来修改' (Select Blog to modify). Below this, there is a table with one row containing a checkbox, the text 'this is a test', and a '1 博客' (1 Blog) label. Above the table, there is a '动作' (Action) dropdown menu with a checked item '删除所选的 博客' (Delete selected blog) and a '执行' (Execute) button. The status '1 个中 0 个被选' (1 of 0 selected) is shown next to the button.

图 5-8 Django 博客

在这个页面里，我们可以删除博客，并且可以在单击标题后进入博客来修改内容。通过 Admin 组件可以将删除（Delete）、修改（Update）、添加（Create）这些内容交给用户后台来做。当然，它也不需要 View/Template 层来做，我们只需要关心如何显示这些数据。

现在可以执行一次新的代码提交：

```
git add .  
git commit -m "create blog admin page"
```

在完成数据的创建、更新和删除之后，就可以转而关注如何显示数据。

5.2.2 在页面上显示博客

在上一节里，我们已经创建了一些数据，现在需要在页面上显示这些数据——从 Model 中找到对应的数据，再返回到前端模板中。与创建首页的步骤类似，具体如下。

①编写 View 逻辑。

②配置好博客应用对应的 URL。

③测试博客的路由是否正确。

④创建前端页面。

⑤测试从首页跳转某个具体的页面。

在绝大多数的 Web 应用中，数据的显示拥有相对固定的模式，即：列表页和详情页。诸如搜索引擎、博客、论坛、新闻、购物、社交网站等，我们都会先到一个列表页，当单击相应的标题时，都会进入相关的页面详情里。

（1）列表页

在列表页里，我们会以一定的关系来显示所有相关的数据，如所有的博客、某一个月里的所有博客、某一主题的所有新闻等。通常只会在这个页面上显示每个数据的部分内容。

如，对博客、新闻应用来说，我们会在列表页上显示摘要、标题、时间等内容；对论坛应用来说，我们显示一个个的标题、时间、作者等内容。一般来说，会存在两种不同的数据源模型。

- 同一数据模型。在这样的应用里，我们从同一个数据模型里获取列表中我们需要的数据。
- 由不同的数据源组成。典型的应用就是搜索引擎、社交网站以及聚合网站等，其列表页需要以某一特定的规则从不同的网站、用户获取，再以一定的形式来格式化这些数据。

在博客应用里只显示博客，因此，只需要从博客的数据源里取出这些数据即可。

(2) 详情页

在详情页，我们会从指定的数据源里取出某一个特定的数据，并展示其需要显示的内容。当用户从列表页到达详情页时，会以 URL 的形式来传递参数，而这个参数是唯一的标识。如之前创建的博客模型，其中的 slug 和 title 都是唯一的。在创建博客的时候，我们还会在后台为其生成一个特有的 ID。

根据这个特定的标识，从我们的数据中获取指定的博客。

1. 创建列表页

依据我们的需求，可以很容易地写出列表页的逻辑代码，如下：

```
from django.shortcuts import render_to_response, get_object_or_404

from blog.models import Blog


def blog_list(request):
    return render_to_response('blog/list.html', {
        'blogs': Blog.objects.all()
    })
```

这里的代码和之前的首页是类似的，我们渲染并返回 `blog/list.html` 文件，稍有不同的是：我们从 `Blog` 模型里取出所有的数据，并将它赋予 `blogs` 对象里。这样，我们就可以在 `blog/list.html` 文件获取博客的所有内容。相应地，我们也可以看看 HTML 文件应该怎么编写：

```
<html>
  <head>
    <title>Growth Studio - Blog</title>
  </head>
  <body>
    {% for blog in blogs %}
      <h2>{{ blog.title }}</h2>
      <p>{{blog.body | slice:"":80}}</p>
      <p>{{blog.posted}} - By {{blog.author}}</p>
    {% endfor %}
  </body>
</html>
```

在上面的代码里，我们使用了 Django 的模板来生成博客的内容。遍历了 `blog_list` 方法中返回的 `blogs` 对象，取出每个博客中的 `title`、`body`、`posted`、`author` 字段的值，然后渲染成 HTML。

你应该注意到了上面的 `body` 字段有些特别，即 `{{blog.body | slice:"":80}}`，在这里使用了一个 Django 的过滤器对文章的长度进行简单切分——只取出前 80 个字符。Django 的模板引擎提供了一系列丰富的过滤器，我们可以在模板文件中对数据进行一些特殊处理。

接着，在 `urls.py` 里添加相应的 route 来访问页面：

```
...
from blog.views import blog_list

urlpatterns = [
    url(r'^$', index),
```

```

url(r'^admin/', admin.site.urls),
url(r'^blog/$', blog_list)
]

```

运行服务，就可以看到如图 5-9 所示的页面。



图 5-9 博客列表页示意图

现在，让我们再写一个测试来确保功能都能测试覆盖到。这次与之前稍微有些不同的是，我们写的单元测试将会覆盖博客的内容，而功能测试则会测试用户可以在页面间跳转。如下是单元测试的代码：

```

class BlogpostListTest(TestCase):
    def setUp(self):
        self.user = User.objects.create_user(username='phodal', email=
            'h@phodal.com', password='phodal')

    def test_blog_list_page(self):
        Blog.objects.create(title='hello', author=self.user, slug='this_
            is_a_test', body='This is a blog',
                                posted=datetime.now)
        response = self.client.get('/blog/')
        self.assertIn(b'This is a blog', response.content)

```

我们在 `setUp` 阶段创建了一个用户并将值赋予 `self.user`，以便在测试阶段来创建博客——创建博客的时候，需要使用到这个用户，接着获取这个页面返回的 HTML，并断言 HTML 中有我们的文章内容。

对应的, 我们也需要在功能测试中的 `setUp` 使用相同的方式来创建这个用户, 这样才能成功创建一篇博客。

```
class HomepageTestCase(StaticLiveServerTestCase):
    def setUp(self):
        self.selenium = webdriver.Chrome()
        self.selenium.maximize_window()
        self.user = User.objects.create_user(username='phodal', email=
'h@phodal.com', password='phodal')
        super(HomepageTestCase, self).setUp()

    def tearDown(self):
        self.selenium.quit()
        super(HomepageTestCase, self).tearDown()

    def test_should_goto_blog_page_from_homepage(self):
        Blog.objects.create(title='hello', author=self.user, slug='this_
is_a_test', body='This is blog detail',
                             posted=datetime.now)
        self.selenium.get(
            '%s%s' % (self.live_server_url, "/")
        )
        self.selenium.find_element_by_link_text('博客').click()

        self.assertIn("This is blog detail",
            self.selenium.page_source)
```

同样, 我们也在测试里创建了一篇博客, 然后打开首页, 并使用 `find_element_by_link_text` 方法找到页面上的“博客”两个字, 再次单击这个链接, 页面将跳转到博客列表页, 最后再断言页面中包含有“This is blog detail”字样。

2. 创建详情页

有了上面的经验, 要创建博客的详情页就会轻松一点。稍有不同的是, 我们需要先进

行 URL 的设计，如下是 URL 配置代码：

```
url(r'^blog/(?P<slug>[^\.]+)\.html', blog_detail, name='blog_view'),
```

我们使用了一个看上去稍微有点复杂的正则表达式 `r'^blog/(?P<slug>[^\.]+)\.html`，它会将形如 `blog/hello-world.html` 中的 `hello-world` 提取出来作为参数传给 `blog_detail` 方法。在 Python 中可以使用 `(?P<name>正则表达式)` 的形式来捕获字符串的值，并将这个值赋予 `name` 变量。

即，我们将使用 `[^\.]+` 来获取 URL 中的链接，再将这个值传给 `blog_detail` 方法来处理：

```
def blog_detail(request, slug):
    return render_to_response('blog/detail.html', {
        'post': get_object_or_404(Blog, slug=slug)
    })
```

上面的代码使用 `get_object_or_404` 从数据库中查询 `slug` 等于传入的 `slug` 的值。如果存在，就返回这篇博客，否则就向页面返回一个 404。

现在，就差 `detail.html` 文件了：

```
<html>
  <head>
    <title>{{ blog.title }}</title>
  </head>
  <body>
    <h2>{{ blog.title }}</h2>
    <p>{{ blog.body }}</p>
    <p>{{ blog.posted }} - By {{ blog.author }}</p>
  </body>
</html>
```

详情页 HTML 的内容比列表页的 HTML 稍微简单一些，唯一不同的是：将博客的标题放在 `title` 标题里，即显示在标题栏上。

详情页的测试与列表页是差不多的，读者可以直接看配套的代码。但是我们可以测试一些异常的情况，即用户访问的是一个不存在的博客时：

```
def test_not_found_blog(self):
    response = self.client.get('/blog/this_not_a_blog.html')
    self.assertEqual(404, response.status_code)
```

如上代码所示，当我们访问一个不存在的博客时，页面返回的 HTTP 状态码应该等于 404。

3. 模板复用

现在，让我们再做一些博客页面的美化工作，做一些小小的重构来复用首页的模板。不过在那之前，让我们按照 Django 的习惯来优化静态文件的路径。

(1) 使用 static 标签优化路径

在前面的内容中，我们通过在配置文件 settings.py 中的 STATICFILES_DIRS 来指定好静态文件的目录。在 HTML 里指出了静态文件的相对位置，如 static/css/carousel.css，当我们在产品环境使用的时候，这些路径应该发生一些相应的变化。因此，我们要使用 Django 中的 static 标签来指明静态文件的相对位置。

在代码里，需要在 HTML 文件的最上方加载 static 标签，并修改对应的 CSS 和 JS 文件的语法，代码如下：

```
{% load static %}
...
<link href='{% static "css/carousel.css" %}' rel="stylesheet">
...
```

(2) 模板共用

在首页已经开发了一套基本的模板，为了在博客的详情页和列表页使用这个模板，需要代码块 block 代码进行重构。我们再 Tasking 一下：

- 抽取出一个基本的模板文件 `base.html`。
- 将首页里的内容提取到 `index.html` 中。
- 基于模板优化列表页和详情页。

下面先以博客详情页为例，来说说这个提取的过程：

```
<html>
  <head>
    <title>{{ blog.title }}</title>
  </head>
  <body>
    <h2>{{ blog.title }}</h2>
    <p>{{ blog.body }}</p>
    <p>{{ blog.posted }} - By {{ blog.author }}</p>
  </body>
</html>
```

我们可以将上面的文件提取成两部分：`blog_list.html` 和 `blog_base.html`。先在 `blog_base.html` 文件中引入 `block`，代码如下：

```
<html>
  <head>
    <title>{{ blog.title }}</title>
  </head>
  <body>
    {% block content %}
    {% endblock %}
  </body>
</html>
```

对应的，我们只可以在 `blog_list.html` 文件中使用 `extends` 标签来扩展这个代码块：

```
{% extends 'blog_base.html' %}
```

```
{% block content %}
    <h2>{{ blog.title }}</h2>
    <p>{{ blog.body }}</p>
    <p>{{ blog.posted }} - By {{ blog.author }}</p>
{% endblock %}
```

同理，对应于在首页里使用的模板，我们只需要保留原来的 Header 和 Footer，并提取为 base.html 文件。部分代码如下：

```
<body>

<div class="navbar-wrapper">
    <div class="container">
        ...
    </div>
</div>

{% block content %}
{% endblock %}

<!-- FOOTER -->
<footer>
    ...
</footer>
```

我们就可以在列表页和详情页共用这个模板，如下是共用模板后列表页的代码：

```
{% extends 'base.html' %}

{% block content %}
<div class="container blog-list">
    <div class="row">
        {% if blogs %}
```

```
{% for blog in blogs %}
<div class="col-sm-4">
    <h2><a
href="{{ blog.get_absolute_url }}">{{ blog.title }}</a></h2>
    {{blog.body | slice:"":80}}
    {{blog.posted}} - By {{blog.author}}
</div>
{% endfor %}
{% else %}
<p>当前还没有博客</p>
{% endif %}
</div>
</div>
{% endblock %}
```

由于这部分内容比较简单，这里就不再详细展开，读者可以直接通过阅读代码来理解。

5.3 数据与 Web 应用开发

或许你发现了，在创建这个博客系统的过程中，我们关注的点是创建、更新、删除、读取，以及如何显示数据。而这也是 Web 应用的核心所在，对于不同的开发人员来说，有不同的关注点：

- 对后端开发人员来说，他们关注如何以正确、可靠的方式来管理数据。
- 对前端开发人员来说，他们关注如何提供一个好的交互界面。
- 对业务分析员来说，他们关注如何分析数据，以便做出更好的业务决策。

下面看看在这个过程中有什么不同之处？

5.3.1 管理数据

对于数据的操作，有增加（Create）、读取（Read）、更新（Update）和删除（Delete）四个基本操作，即 CRUD。这些也是关系数据库中所需要支持的主要功能。

在创建数据的时候，我们主要考虑两点：**数据验证和鉴权**。首先，需要对用户进行权限鉴定，以判断当前的用户是否能创建该数据。随后用户在页面上提交数据，这时就需要对数据进行验证。我们需要知道数据是否符合要求，如是否是唯一的值、是否超过设置的长度、是否符合数据规则等。通常也会使用 JavaScript 代码来实时校验数据是否符合要求，以实现更好的用户体验——避免用户在传给服务器端才告知用户数据有问题，再重新提供数据。这种设计不仅体验不好，而且会消耗更多的服务器资源。

为了集中精力完成创建及显示的过程中，我们将如何使用 Django 自带的用户管理模块来提高开发效率。同样也看到了这个过程首先要登录，这个过程就是对用户权限的鉴定；然后创建数据，在这个过程中要对数据进行验证——如标题是不是唯一的。在不同的项目里会稍微有一些区别，但是这个区别并不会太大。

更新数据和创建数据是类似的，先鉴权，再修改。而对删除数据来说，则只需要关注用户是否有删除的权限即可。

对读取数据来说，也会存在着鉴权的操作——当用户只能看到自己的数据，或者看到别的用户允许自己看的数据时。而在博客系统里，则所有的人都可以阅读我们的数据。在这时，我们就不需要对用户的权限进行限定。

上面的这些数据管理主要是针对后台开发来说的。对前端开发来说，则需关心如何显示好数据。

5.3.2 显示数据

在编写博客系统的过程中，后台用对象的形式将数据返回给前端模板。在前端模板里，只选择了 title、body、author 等字段来显示这些数据，并对 body 字段进行过滤。最后编写

一些 HTML 和 CSS 来美化界面。因此，我们可以将其归纳为三个步骤：

- 获取数据。
- 过滤数据。
- 美化界面。

对采用后台模板形式来返回 HTML 的应用来说，HTML 都是在服务器完成渲染及过滤的，最后再由浏览器显示在用户页面上。

对单页面应用来说，则由前台向后台来发起数据请求，通常是 Ajax 请求或者 Fetch 的方式。出于性能或者便捷考虑，服务器可能会直接返回未过滤的数据，这时由前台来对数据进行过滤——选择需要的数据字段，并对一些特殊的字段进行处理；最后再将修改变动到浏览器上。

5.4 小结

本章仍然结合了精益与小步前进的方式来展示内容，每一小步都有一个可以工作的原型，这个原型就是对成功进入下一步的反馈。我们介绍了如何使用 Django 框架来创建着陆页；随后创建了博客系统的模型，并结合 Django 后台创建了一个简单的博客系统。我们穿插了如何编写不同级别的测试，以保证功能不会随着功能的增加而被破坏。

虽然，我们只使用了一章的篇幅来介绍使用 Django 创建博客应用，但这些都需多加练习才能掌握好。在这个过程里，我们要练习的主要是思维的转变。在实际的编程过程中，首先是要有我们的想法，其次才是行动——想法决定了我们下一步的方向。

1. 编程如同写作

编程这件事情实际上一点儿也不难，当我们只是在使用一个工具创造一些东西的时候，比如拿着电烙铁、芯片、电线等去焊一个电路板的时候，我们学的是如何运用这些工具。虽然最后我们的电路板可以实现相同的功能，但是我们可以一眼看到差距所在。换个

贴切一点的比喻，比如烧菜做饭，对一个优秀的厨师和一个像我这样的门外汉而言，就算给我们相同的食材、厨具，一段时间后也许是诱人的美食，一份只能喂猪了——即使我模仿着厨师的步骤一步步做，也许看上去会差不多，但是一吃便吃出差距。

我们做不好饭，焊不好电路，还写不好代码，很大程度上并不是因为我们比别人笨，而只是别人比我们做得更多。有时候一种机缘巧合的学习或者 Bug 的出现，对不同人的编程人生都会有不一样的影响。我们只是在使用工具，使用得好与坏，在某种程序上决定了我们写出来的质量。写字便是如此，给我们同样的纸和笔，不同的人写出来的字的差距很大，写得好的相比于写得不好的，只是因为练习得更多。而编程难道不也是如此吗？最后写代码这件事就和写字一样简单。

当真正编程的时候，我们还会纠结于“僧推月下门”还是“僧敲月下门”的时候，当我们越来越熟练时，就容易决定究竟用哪一个更合适。而这样的“推敲”，无论在写作中还是在编程中都是相似的过程。

2. 扩展练习建议

①创建一个博客。

②美化博客系统的 UI。

②依据不同的时间来创建不同的列表页。

④添加博客的搜索功能。

3. 书籍建议

由于国内在软件工程实践中的测试上多有不足，读者如果对编写测试有兴趣，可以阅读下面的两本书。

- 《Python Web 开发：测试驱动开发方法》，这本书也是以 Django 框架为基础介绍如何开发 Web 应用。书的前半部分以测试驱动开发的方法来介绍如何开发 Web 应用；后半部分则关注于持续交付等话题。作者在书中对实践过程中遇到的问题进行了一些总结——如测试速度、拆分测试、何时使用集成测试。

- 《优质代码——软件测试的原则、实践与模式》，这本书深入介绍了编写软件测试之后会遇到的一些问题——如何正确实现设计意图，还介绍了测试的力度、分离关注点、减少耦合等测试原则，并介绍了在不同的情形下应该如何编写测试。书中的原则都结合了一定的示例代码，代码是以 Java 语言实现的，但也适用于其他语言。

第 6 章

上线

MVP 即最简可行产品，它不仅体现在产品设计的过程中，还应该体现在开发的过程中。在我们完成了核心的基本功能之后，就需要考虑上线的事情。上线应用时，需要学会如何手动部署。本章将介绍 LNMP 架构，以及如何进行部署，并介绍如何使我们的系统是可配置的，以及如何自动化部署应用。

部署策略是我们在创建完 Demo 之后，要开始着手做的事情。本章将先介绍如何以手动的方式来部署应用到服务器，然后介绍如何使用 Fabric 来实现这部分的自动化部署。实际上，手动部署与自动化部署的步骤是差不多的：

- 安装运行时所需要的软件包。
- 获取代码或者已编译的二进制包。
- 运行我们的应用。
- 为应用配置地址、缓存等。

当我们手动部署应用时，流程本身是固化的。我们很容易将这个步骤变成自动化，只是需要保证拥有一个相同的运行环境。这时就需要使用代码或者工具将我们的环境与代码隔离。

6.1 手动部署

在部署之前，让我们回顾一下服务器以及运行在上面的软件。你可能已经在不同的场合听说过 LNMP——由几个单词组成的缩写，即：

- L，即 Linux，对应于操作系统。
- N，即 Nginx，对应于 HTTP 服务器。
- M，即 MySQL，对应于数据库。
- P，即 PHP / Python 等，对应于编程语言。

这几个字母可以让我们对服务器有一个大致的了解。我们在第 2 章中提到，当我们访问一个网站时，在某种意义上可以看作是访问一个远程的计算机。我们访问这个计算机的 80 或者 443 端口，操作系统先捕获这个请求，再交由相应的绑定端口的软件请求来处理，即 HTTP 服务器。HTTP 服务器根据其访问的域名，再将请求交给相应的应用来处理。

如在第5章中提到的 Django 框架，Django 再根据 URL 将请求交由应用中的某个函数来处理。函数再依据请求的需要，在数据库中执行操作来获取所需要的数据。最后这些数据将被渲染成 HTML，这个 HTML 将返回给浏览器，并由浏览器来渲染页面。

在这个过程中，我们需要四个主要的软件功能，即操作系统、服务器软件、数据存储软件和应用。部署的过程就是在寻找一种正确的方法让几个软件串在一起。

6.1.1 操作系统与服务器软件

当蒂姆·伯纳斯发明万维网的时候，他也编写了世界上第一个 HTTP 服务器 CERN HTTPd。当时这个服务器运行在一个 NeXT 计算机上，这个计算机当时运行的是一个基于 UNIX 的操作系统 NeXTSTEP。虽然当时的第一个网页只是拥有简单的 HTML，但是它们已经有了现代网页的雏形。为了成功运行起我们的应用，我们还需要一个合适的操作系统、一个服务器软件，以及一个用于生产环境的数据库。因此，让我们简单做个技术选型吧。

1. 操作系统与服务器软件的选择

(1) 操作系统的选择

最早的操作系统、服务器软件、浏览器都运行在 NeXT 计算机上，后来它被移植到其他 UNIX 系统上。随后，微软也在 Windows 上推出了自己的 HTTP 服务器 IIS。UNIX 操作系统在最初的时候由于其开放性而深受市场欢迎，而在后来遇到一系列的版权问题流失了相当多的用户。在今天 UNIX 操作系统主要与中大型服务器一起使用，更多的是面向企业用户。与此同时，大量的用户和开发者将目光投向其他方向——使用 GNU 协议开源的 Linux 内核的操作系统。Linux 内核本身是开源且免费的，大量的开发者围绕这个内核构建出了一些受欢迎的操作系统，如 Ubuntu、OpenSuSE、Debian 等，并且我们可以发现现有主流的云服务提供商（如 Amazon AWS、阿里云、青云等）提供的云服务主要都是基于 GNU/Linux 操作系统。

在今天有近一半的服务器运行在基于 GNU/Linux 内核的操作系统之上。在服务领域

比较受欢迎的 GNU/Linux 操作系统是 CentOS，在桌面领域比较受欢迎的 GNU/Linux 操作系统是 Ubuntu。考虑到使用及学习的便捷性，本文将介绍如何在 Ubuntu 上运行我们的应用程序。在掌握了如何使用 Ubuntu 部署后，可以用 Ubuntu、Debian 或者 CentOS 来部署应用。本书在写作过程中使用的是虚拟机来进行部署，虚拟机提供了一种快照机制——类似于版本管理的机制，可以让我在出错时退回到上次的修改。读者可以在学习时使用虚拟机或者 Raspberry Pi 这一类微型计算机来模拟服务器。

事实上，当我们选择了一门语言的时候，也决定了使用哪个操作系统——尽管我们可以在 Windows 上运行 Python 语言，但是它不能发挥出它最好的性能。当我们决定使用 .NET 技术来开发 Web 应用时，最好的选择就是用 Windows Server；当我们决定使用 Ruby、PHP、Python、Java 等语言时，选择使用 Linux 服务器。这主要是由于 Microsoft 在当时没有将 .NET 技术移植到 GNU/Linux 系统上。而最近几年里，Microsoft 正在走向开源，相信在未来使用 .NET 技术时也可以使用 Linux 服务器。

在开发时，我们可以使用不同的操作系统（Windows、MacOS、Ubuntu 等），在部署时用 GNU/Linux 操作系统。

（2）服务器软件选择

在蒂姆·伯纳斯编写一个版本的 HTTP 服务器 CERN HTTPd 之后，位于美国伊利诺伊大学香槟分校的国家超级电脑应用中心（简称 NCSA）也开发出了自己的 HTTP 服务器，被称之为 NCSA HTTPd。在这个服务器软件里拥有一个名为通用网关的接口（CGI），它是一个独立于编程语言的接口，可以让开发者使用不同的服务器来实现动态的网页。当编程语言接收到来自 CGI 的请求时，就可以依据需要从数据库中提供内容。由于 NCSA HTTPd 本身是开源的，现在流行的 Apache 服务器就是基于 NCSA HTTPd 之上开发的。

通过学习上面的内容，我们对 HTTP 服务有了一个大概了解。HTTP 服务器与编程语言之间是独立的，这就意味着：我们可以在一个服务器上使用不同的编程语言，运行不同的 HTTP 服务，并运行着不同的网站。当我们决定使用 Linux 时，就只能从 Apache 和 Nginx 中选择一。

我们选择 Nginx 的主要原因是：对静态文件处理比较好，更适合 Django。同时，在

最近几年里, Nginx 服务器越来越受市场欢迎, 有相当多的教程及书籍。

2. 部署第一个页面

在服务器上使用命令来执行操作与在自己的计算机上执行并没有太大的区别——除了网络质量可能会影响操作体验。在 Windows 机器上使用鼠标和键盘进行操作, 在 Windows 服务器上同样也是如此, 需要安装远程桌面的软件才能用客户端来访问服务器端。在 Linux 机器上使用命令行来安装软件、执行脚本等, 在 Linux 服务器上也是如此, 需要安装相应的通信软件, 即 OpenSSH (OpenSSH 的全称是 OpenBSD Secure Shell), 它是安全 Shell 协议族 (SSH) 的一个免费版本, 用于远程控制和文件传输。与 Telnet 的明文传输相比, OpenSSH 提供了服务器端及客户端工具, 用于加密远程控件和文件传输过程中的数据。

对我们购买的服务器来说, 它们已经在服务器端安装好这个软件, 否则将无法访问这些服务器。通常来说, 我们所使用的 GNU/Linux 操作系统也安装了相应的客户端工具。不过, 如果想把计算机当成服务器, 就需要在计算机上安装服务器端的工具, 如:

```
sudo apt-get install openssh-server
```

那么, 我们就可以使用相应的 SSH 客户端来访问服务器。如在 Ubuntu 下, 可以使用 ssh 命令来访问机器:

```
ssh phodal@192.168.4.12
```

Windows 用户也可以使用 PuTTY 软件来做同样的事。当我们登录服务器后, 就可以看到最近相应的一些服务器信息, 并直接进入命令行模式。

```
The authenticity of host '99.12.195.84 (99.12.195.84)' can't be established.  
ECDSA key fingerprint is SHA256:CloJQOJPwQGGSh8nUXDGVtjPlpGqV+DEXrnNWhzwiqI.  
Are you sure you want to continue connecting (yes/no)? yes  
Warning: Permanently added '99.12.195.84' (ECDSA) to the list of known hosts.  
phodal@99.12.195.84's password:  
Welcome to Ubuntu 16.10 (GNU/Linux 4.8.0-22-generic i686)
```

```
* Documentation: https://help.ubuntu.com
* Management:   https://landscape.canonical.com
* Support:       https://ubuntu.com/advantage
```

9 个可升级软件包。

8 个安全更新。

The programs included with the Ubuntu system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by
applicable law.

```
phodal@ubuntu:~$
```

随后我们就可以像操作正常的计算机一样操作服务器，让我们先安装 Nginx 来作为 HTTP 服务器：

```
sudo apt-get install nginx
```

安装完后，就可以使用 IP 访问网站，页面如图 6-1 所示。

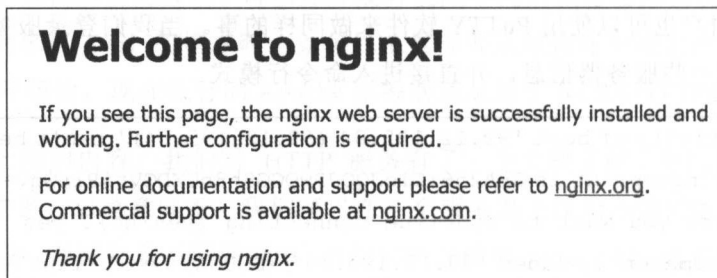


图 6-1 Nginx hello, world

为了使用域名，诸如 `https://www.phodal.com` 的形式来访问我们的网站，我们还需要在 DNS 服务器上配置好 IP 与域名的关系——通过访问域名服务提供商来进行相应的操作，在相应的域名下添加一个新的记录，如图 6-2 所示。

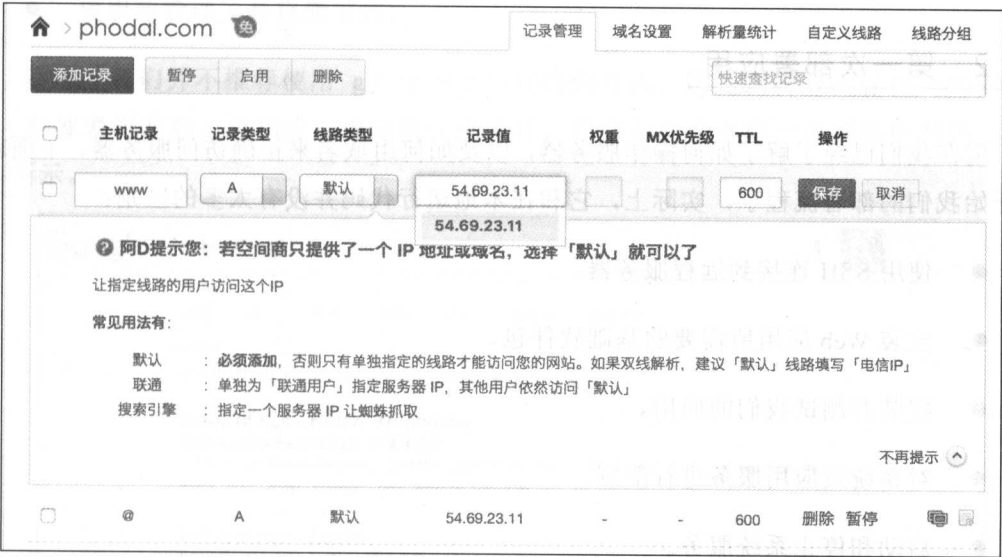


图 6-2 DNS 解析设置示例

我们只需要添加域名前缀（即图 6-2 中的主机记录）和服务器的 IP 地址（即图 6-2 中的记录值）。

配置完成后，DNS 服务器已经指向了我们的服务器，我们只需要修改 `/etc/nginx/sites-available/default` 文件，将其中的 `default_server` 改成域名即可，如：

```
server {  
    listen 80 www.phodal.com;  
    listen [::]:80 www.phodal.com;  
    ...  
}
```

修改完后，重启 `nginx` 服务，就可以用域名来访问服务器了。

```
systemctl restart nginx
```

不过通常来说，我们会使用 `sudo nginx -t` 来测试配置文件是否正确，随后再重启服务。

6.1.2 第一次部署应用

现在我们已经了解了如何操作服务器，以及如何用域名来正确访问服务器。下面就可以开始我们的部署流程了，实际上，它和在本地运行代码并没有太多的区别：

- 使用 SSH 连接到远程服务器。
- 安装 Web 应用所需要的基础软件包。
- 安装并测试我们的应用。
- 对系统及应用服务进行配置。
- 启动和停止系统服务。

当我们在第一次配置的时候，要先让服务可以在服务器上运行起来。即，安装好应用所需要的基础环境，安装的软件包在这个过程会依据需要进行安装，该过程只要保证使用 `python manage.py runserver` 正确运行即可。再使用 Nginx 以及 Gunicorn 来提供服务质量，并保证服务的可用性。

在第一次部署的时候，我们应该尽可能地记住部署的流程。自动化部署的步骤是依赖于这一步步积累而成的，并且配置也依赖于这个步骤的配置。因此，要尽可能地保证它们是可以重复进行的。

1. 在服务器上运行 runserver

在这一步，我们要做的就是让应用在服务器上运行。因此，首先需要将软件包放置到服务器上，方法有多种，例如：

- 使用 git 将代码克隆到服务器上。

- 使用 scp 上传软件包。
- 从某个服务器上下载软件包。
- 使用包管理工具直接安装。

不过，我们并不推荐使用 git 作为上传代码的方式。因为这种方式存在一些安全隐患，在搜索浏览器上对相应的内容进行搜索时，很容易就会发现一些网站的源码，如图 6-3 所示。



图 6-3 Git 管理问题

虽然这里使用的是代理来转发请求，并且分配了静态文件的路径——用户无法直接使用 .git 目录访问我们的文件夹，并且当我们使用 git 的时候，需要在服务器上配置私钥，当服务器被攻破时，导致代码库泄漏——黑客不仅可以访问这个项目的代码，还可以访问其他项目的代码。

在这里要使用的方式是：基于 GitHub 的标签功能来管理版本。我们只需要打个新的标签（Tag），然后就可以从 GitHub 上获取这个版本的代码。

因此，在服务器上可以使用 `wget` 来获取某个版本的软件，如 0.0.3: `wget https://codeload.github.com/phodal/growth_studio/tar.gz/v0.0.3`，接着解压这个压缩包，就可以得到用于部署的代码：

```
tar -xvf v0.0.3.tar.gz
```

剩下的就是安装依赖并创建虚拟环境等操作。与第 3 章中提到的类似，首先需要查看 Python3 的版本：

```
python3 --version
```

并检测是否已经安装好 `pip3`。如果没有，可以使用下面的命令来安装 `pip3`：

```
sudo apt install python3-pip
```

然后安装 `virtualenv`，并使用它来创建虚拟环境。接着安装应用的依赖 `pip install -r requirements/prod.txt`，或者安装 `Fabric3`，然后用 `fab install`。

随后做数据迁移：`python manage.py migrate`。最后，就可以运行服务：`python manage.py runserver`。

现在，已经可以直接在我们的开发机器上访问了——服务器的 IP + 8000 端口，如 192.168.4.12:8000。如果无法直接访问这个端口，则可能需要使用 `ufw` 或者 `iptables` 来配置相应的端口权限。

2. 使用 Gunicorn 进行多进程

虽然在上一节里已经启动了 Web 应用服务。但是它并不能直接用于生产环境，这不安全，并且会降低程序运行效率。

- 在开发环境中，Django 自带了一个简单的 WSGI 服务器来运行应用，这个 WSGI 服务器缺少一些在产品运行时的高级功能，如多进程、进行自动优化等。

- 使用 Django 来处理静态文件，这并没有什么问题。而当我们在线上使用 Django 来处理这些静态文件请求的时候，将会加重应用的负载，这部分内容应该交由 HTTP 服务器来处理。
- 当程序在运行过程中遇到一些 Bug，则可能就因此而退出，此时需要一个工具来重启应用。
- Django 自带的 WSGI 服务器只支持单线程，而在线上时使用多线程可以同时进行更多的处理。

基于上述因素，除了已经安装好的 Nginx，还需要额外的两个工具来帮助我们完成线上的部署：

- WSGI 服务器——Gunicorn。
- 进程守护工具——Supervisor。

Gunicorn 是一个面向类 UNIX 操作系统的开源 Python WSGI HTTP 服务器，它由 Ruby 语言里的 Unicorn 移植而来，采用 pre-fork 模式。它可以兼容大部分 Web 框架，并且集成方便，对服务器资源消耗少，而且运行起来相当快。Gunicorn 本身使用 Python 语言来编写，因此，仍然可以直接由 pip 来安装这个软件：

```
pip install gunicorn
```

现在让我们使用 Ctrl+C 组合键中断之前的 Web 服务，然后使用下面的命令来运行 Web 应用。

```
$ gunicorn growth_studio.wsgi
```

```
[2016-11-16 22:26:37 +0800] [70141] [INFO] Starting gunicorn 19.6.0
[2016-11-16 22:26:37 +0800] [70141] [INFO] Listening at:
http://127.0.0.1:8000 (70141)
[2016-11-16 22:26:37 +0800] [70141] [INFO] Using worker: sync
```

```
[2016-11-16 22:26:37 +0800] [70144] [INFO] Booting worker with pid: 70144
```

在上面的命令里，我们向 Gunicorn 传入了应用的 WSGI 模块——即 Web 服务器网关（Python Web Server Gateway Interface），它是为 Python 语言定义的 Web 服务器和 Web 应用程序或框架之间的一种简单而通用的接口。它基于上面提到的 CGI 标准而设计。在不同的语言里都有类似的机制，如 Java 里的 Servlet、PHP 语言里的 FastCGI 等，其工作原理都是类似的。

WSGI 由两部分配合完成：一部分是 Web 服务器部分，如 Nginx 或者这里的 Gunicorn，另一部分则是应用程序部分，即这里的代码。当一个新的请求到来时，WSGI 服务器部分将执行应用程序，并为应用程序部分服务环境变量信息以及一个回调函数。应用程序部分将处理这个请求，并使用服务器部分提供的回调函数返回结果。

在运行 `startproject` 创建项目的时候，将会生成一个默认的 WSGI 配置文件，如项目中的 `growth_studio/wsgi.py`，其内容如下：

```
import os
from django.core.wsgi import get_wsgi_application

os.environ.setdefault("DJANGO_SETTINGS_MODULE", "growth_studio.settings")

application = get_wsgi_application()
```

当我们使用 WSGI 服务器调用应用的时候，将会在 `os.environ.setdefault` 中设置应用的配置文件路径。WSGI 服务器加载应用时，将导入 `settings` 模块。在 `get_wsgi_application` 方法里初始化应用，并在其中处理对应的 URL 请求，最后返回给 WSGI 服务器。

这时我们就会遇到一个很有趣的事情，即博客中的静态文件（即诸如 CSS、JavaScript 和图片等资源文件）都消失了，如图 6-4 所示。

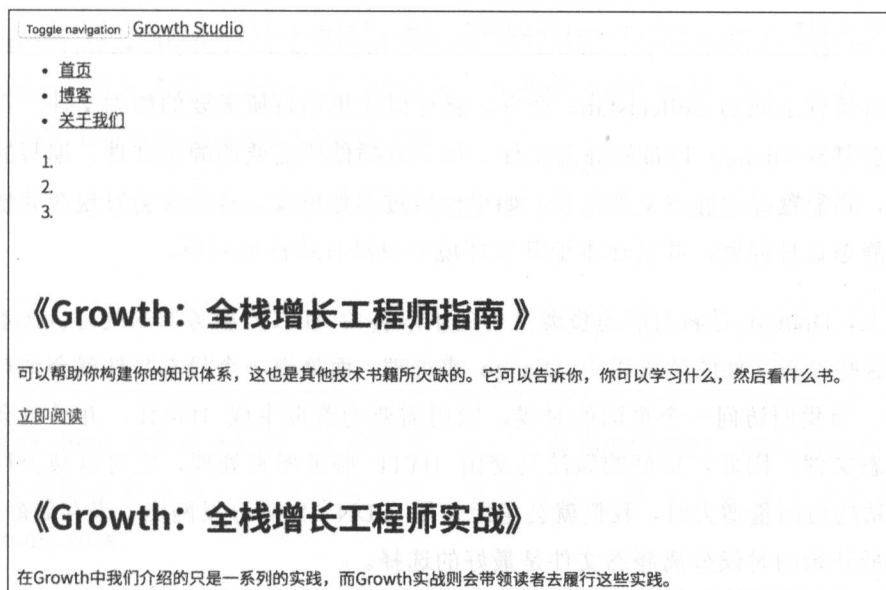


图 6-4 缺少静态文件

先让我们修复一下这个问题，再来看看导致这个问题的原因。Django 框架中有一个模块是 `staticfiles`，它提供了一个用于管理这些静态文件的命令，即：

```
python manage.py collectstatic
```

这个命令收集这个项目所使用到的静态文件到指定的目录。因此，如果没有配置相应的静态目录，就会遇到下面的错误：

```
django.core.exceptions.ImproperlyConfigured: You're using the staticfiles app without having set the STATIC_ROOT setting to a filesystem path.
```

这时就需要在 `STATIC_ROOT` 配置里添加静态文件配置所需要的路径：

```
STATIC_ROOT = '/var/www/growth-studio/static/'
```

或者，也可以使用相对于项目的路径：

```
PROJECT_DIR = os.path.dirname(os.path.abspath(__file__))
```

```
STATIC_ROOT = os.path.join(PROJECT_DIR, 'static')
```

随后再运行上面的 `collectstatic` 命令，就可以收集项目所需要的静态文件。在这个过程中，它会复制 Django 自带的静态文件、第三方插件所需要的静态文件、编写的那些静态文件等，到配置好的静态文件路径。如果你用过其他框架，可能就会发现在其他框架里很少遇到静态这种问题，并且在本地开发环境中也没有这样的问题。

实际上，Django 是将对静态资源文件的请求转由 HTTP 服务器来处理。在本地开发的时候，这些静态文件的请求将由 Django 来处理，而处理一个静态文件就会消耗应用程序的资源。当我们访问一个页面的时候，应用需要为首页生成 HTML，并且也会访问相当多的静态文件。因此，更好的做法是交由 HTTP 服务器来处理，它可以减少程序的瓶颈。当网站的访问量增大时，我们就会考虑使用 CDN（内容分发网络）来存储静态文件。因此，在最开始的时候分离静态文件是最好的选择。

如果在中途退出了服务器，你可能会发现应用已经停止服务。当我们登录服务器的时候相当于创建了一个新的会话，结束会话的时候，系统会 HUP（hangup）信号，从而关闭其所有的子进程。这时我们在服务器上运行的应用服务就会被停止，因此可以先通过 `nohup` 来保持应用在后台运行。`nohup` 顾名思义就是让提交的命令忽略 `hangup` 信号。使用 `nohup` 时，只需要在命令前添加 `nohup`，在命令结尾的地址添加一个 `&` 符号，代码如下：

```
nohup gunicorn --workers=2 growth_studio.wsgi --bind 0.0.0.0:8000&
```

这里的 `workers=2` 参数的意思是使用两个 `worker` 进程，即系统可以处理两个并发请求。随后我们将介绍 `supervisor`，它更适合管理应用的进程。

3. 使用 Nginx 处理静态文件

Gunicorn 仍然不是最好的选择。为了提供更好的性能，我们还需要使用 Nginx 来处理静态文件请求，并可以借助 Nginx 优异的性能来提供页面缓存服务。同时，还可以使用 Nginx 来承担安全任务，如反爬虫、限制 IP 访问等。因此，这里使用 Gunicorn 来提供应用的多线程服务，用 Nginx 来承担静态文件等请求。使用 Nginx 来反向代理 HTTP 请求给 Gunicorn，由 Gunicorn 再将请求交由 Web 应用程序。

Gunicorn 与 Django 应用之前使用 WSGI 进行通信，Gunicorn 与 Nginx 则使用套接字（socket）进行通信。UNIX 域套接字是一种进程间通信机制（IPC），用于让运行在同一台机器上的进程间进行双向的数据交换。

我们需要先更新 Gunicorn 运行机制，在运行时绑定一个 UNIX 域套接字。运行脚本如下：

```
nohup gunicorn --workers=2 --bind UNIX:/home/phodal/growth-studio/growth-studio.sock growth_studio.wsgi&
```

这里的--bind 参数用于创建一个套接字，出于查看使用的原因，我们将这个文件放置在项目代码所在的位置。如果没有意外，就会在项目目录下生成一个套接字文件 growth-studio.sock。

在那之前，我们可以用 `lsuf -i:8000` 找到正在运行的 Gunicorn 进程：

COMMAND	PID	USER	FD	TYPE	DEVICE	SIZE/OFF	NODE	NAME
Python	62784	fdhuang	5u	IPv4	0x19c0ba61ac031b5			
0t0	TCP	localhost:irdmi	(LISTEN)					
Python	62787	fdhuang	5u	IPv4	0x19c0ba61ac031b5			
0t0	TCP	localhost:irdmi	(LISTEN)					
Python	62788	fdhuang	5u	IPv4	0x19c0ba61ac031b5			
0t0	TCP	localhost:irdmi	(LISTEN)					

并使用 `kill -9 + PID` 来结束进程，如 `kill -9 62784`。由于 Gunicorn 运行了多个进程，因此需要一个个结束。

随后，我们还需要重新配置一个 Nginx，将代理指向套接字文件。我们可以创建一个 `/etc/nginx/sites-available/growth-studio` 文件，其内容如下：

```
server {
    listen 80;

    location / {
```

```
        include proxy_params;
        proxy_pass UNIX:/home/phodal/growth-studio/growth-studio.sock;
    }

    location /static/ {
        root /home/phodal/growth-studio/;
    }
}
```

提示：出于讲解方便，我们只在上面的配置中列出重要的条目，更详细的配置可以从附带代码中获取。

在上面的配置里，`location /`用于匹配所有以 `/` 开头的地址，即匹配到所有的请求。由于 Nginx 采用遵循最长匹配规则，假设一个请求匹配到了两个普通规则，则选择匹配长度大的那个。因此，当浏览器请求的是静态文件时，将会匹配 `/static/` 规则。

因此，在处理非静态文件时，将交由 `proxy_pass` 中指定的套接字文件的路径进行通信。请求静态文件时，将从目录 `/home/phodal/growth-studio/` 中寻找对应的文件。

然后，将配置设置成 Nginx 默认的配置，再重启 Nginx 服务即可。Ubuntu 上的 Nginx 默认使用 `/etc/nginx/sites-enabled/` 作为 Nginx 配置文件存储路径，并使用 `default` 文件作为默认文件。

因此，我们只需要将 `default` 替换掉即可，可以使用下面的代码来创建连接：

```
sudo ln -s /etc/nginx/sites-available/growth-studio /etc/nginx/sites-enabled/default
```

并使用 `systemctl` 命令重启服务。

```
sudo systemctl restart nginx
```

现在，我们就可以正式将这个网站上线了。当然，还可以稍微做出一些改进。

4. 使用管理进程工具

使用 `nohup` 运行我们的程序存在一个问题，即当应用崩溃的时候，用户就无法访问。这时，可以通过一些进程管理工具来重启这个应用。在使用 Python 2.x 版本时，我们使用 `Supervisor`，它是一个使用 Python 2.x 编写的进程管理工具。当使用 `Supervisor` 进行进程管理时，它将监控 `Gunicorn` 服务的状态，如图 6-5 所示。

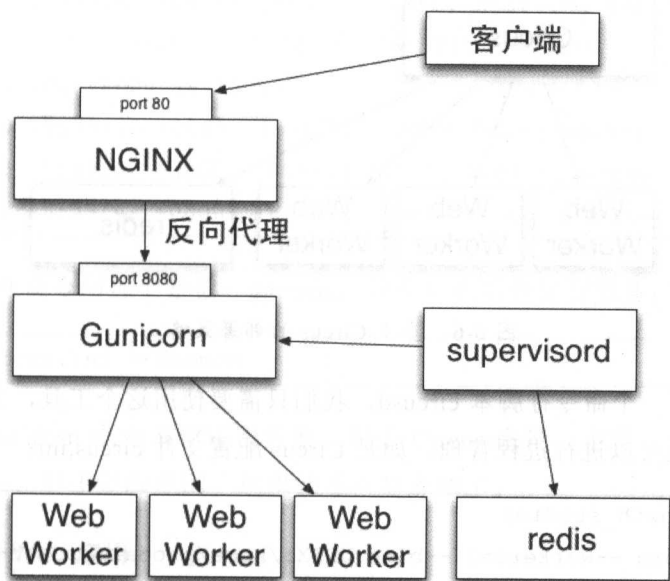


图 6-5 经典的部署

由于这里使用的是 Python 3 作为开发语言，为了方便，我们选择 Python 3 下的进程管理工具 `Circus`，它可以用于监控及控制应用程序的进程及套接字。因此，在使用 `Circus` 时不需要如图 6-6 所示的部署。

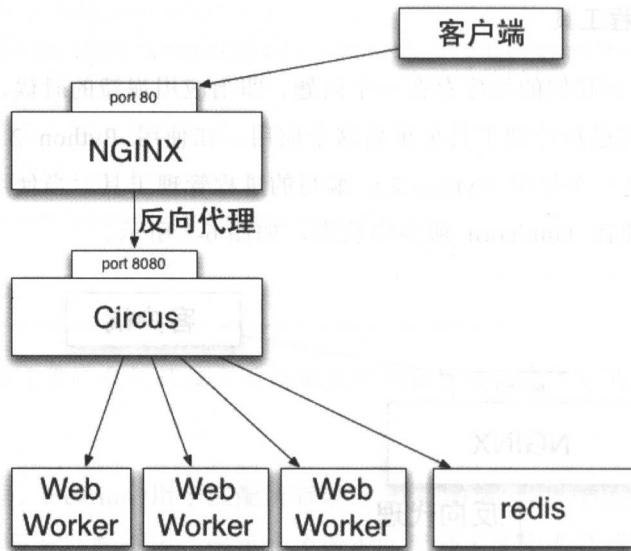


图 6-6 基于 Circus 的部署策略

Circus 提供了一个命令行脚本 `circusd`，我们只需要使用这个工具，并创建一个 ini 风格的配置文件，就可以进行进程管理。如是 Circus 配置文件 `circus.ini`：

```
[watcher:growth_studio]
cmd = gunicorn --workers=2 --bind UNIX:/home/phodal/growth-studio/growth-
studio.sock growth_studio.wsgi
working_dir = /home/phodal/growth-studio
copy_env = True
virtualenv = /home/phodal/py35env
send_hup = true
```

不过，安装 Circus 的方式与之前安装一般软件包的方式稍有不同——需要安装到全局下使用，而不是在创建的 Python 虚拟环境中。如果你已经在虚拟环境中，则可以通过执行 `deactivate` 来退出虚拟环境。现在只需要执行下面的命令，就可以安装 Circus：

```
sudo pip3 install circus
```

然后可以使用 `circusd` 命令加上配置文件运行服务：

```
$ circusd circus.ini

2016-11-19 15:39:41 circus[85573] [INFO] Starting master on pid 85573
2016-11-19 15:39:41 circus[85573] [INFO] Arbiter now waiting for commands
2016-11-19 15:39:41 circus[85573] [INFO] growth_studio started
[2016-11-19 15:39:42 +0800] [85577] [INFO] Starting gunicorn 19.6.0
[2016-11-19 15:39:42 +0800] [85577] [INFO] Listening at:
http://127.0.0.1:8000 (85577)
[2016-11-19 15:39:42 +0800] [85577] [INFO] Using worker: sync
[2016-11-19 15:39:42 +0800] [85582] [INFO] Booting worker with pid: 85582
[2016-11-19 15:39:42 +0800] [85583] [INFO] Booting worker with pid: 85583
```

我们只需要在命令后加上参数 `--daemon`，就可以在后台运行服务：

```
circusd circus.ini --daemon
```

除了使用 Gunicorn 作为 WSGI 服务器，对于 Circus 来说，还有一个不错的 WSGI 服务器是 Chaussette。限于篇幅原因，这里就不重复介绍了。

5. 开机启动 Web 应用

现在已经完成了大部分部署工作，还需要考虑的一点是：如果服务器异常关机，那么开机的时候应用程序也应该是可以运行的。因此，需要设置让应用程序可以开机启动。

我们将使用 Upstart 来执行这个工作，Upstart 是一个用于替代传统 `init` 的系统初始化程序。它会在操作系统中加载内核，挂载根目录等工作后，会调用 `/sbin/init` 来接管后续的服务启动过程。我们只需要创建一个 Upstart 脚本文件 `/etc/init/circus.conf` 即可：

```
description "circusd"

start on filesystem and net-device-up IFACE=lo
stop on runlevel [016]
```

```
exec /usr/local/bin/circusd /etc/circus/circusd.ini
```

代码中的 `start on` 表明服务运行的时机是：在文件系统和本地回环 IP 网络已经准备就绪后再执行。`stop on` 说明服务将会在运行级别 0、1、6（关机、无网络、重启）时停止。最后的 `exec` 才是我们的运行脚本。

6. 部署检查清单

一般的 Web 框架都会搭配有一个部署文档和一些检查清单。前面已经完成了部署部分，下一步就是对部署工具的检查。在 Django 的官网上有一个 `Deployment checklist`（链接地址：<https://docs.djangoproject.com/en/1.10/howto/deployment/checklist/>），用于校对我们的配置是否安全、可靠等。这份检查清单包含下面一些内容：

- 必须设置正确的安全级别。
- 为不同的环境创建不同的配置。
- 允许额外的安全功能。
- 允许性能优化。
- 提交错误报告。

尽管这是 Django 官方的部署检查清单，但它对其他 Web 框架来说也是适用的。Django 提供了一个命令来循环检测应用是否符合这些规范：

```
python manage.py check --deploy
```

接着，就会发现应用存在多达 10 个安全警告：

```
System check identified some issues:
```

```
WARNINGS:
```

```
?: (security.W004) You have not set a value for the SECURE_HSTS_SECONDS
setting. If your entire site is served only over SSL, you may want to consider
setting a value and enabling HTTP Strict Transport Security. Be sure to read
```

the documentation first; enabling HSTS carelessly can cause serious, irreversible problems.

?: (security.W006) Your SECURE_CONTENT_TYPE_NOSNIFF setting is not set to True, so your pages will not be served with an 'x-content-type-options: nosniff' header. You should consider enabling this header to prevent the browser from identifying content types incorrectly.

...

System check identified 10 issues (0 silenced).

因此，我们需要一个个地修复它们。对于大部分内容，我们只需要在 `settings.py` 中将其设置为 `True` 即可。如下是 HTTPS 协议下对 COOKIE 的设置（如果你的网站没有使用 HTTPS，请不要将其设置为 `True`）：

```
SESSION_COOKIE_SECURE = True
CSRF_COOKIE_SECURE = True
CSRF_COOKIE_HTTPONLY = True
```

对于 `DEBUG`，将其设置为 `False`：

```
DEBUG = False
```

以及对应的 HTTPS 相关的设置（如果你的网站没有使用 HTTPS，请不要将其设置为 `True`）：

```
SECURE_HSTS_SECONDS = True
SECURE_SSL_REDIRECT = True
SECURE_CONTENT_TYPE_NOSNIFF = True
SECURE_BROWSER_XSS_FILTER = True
SECURE_HSTS_INCLUDE_SUBDOMAINS = True
```

上面这几个都是关于 HTTPS 和 COOKIE 的安全设置，这里不展开讨论。下面说说两个比较有意思且危险的内容。

①关闭调试模式。在本地开发的时候，`settings.py` 中默认设置 `DEBUG = True`，因此需

要在产品环境将其设置为 `False`。这一点很重要，当我们的应用在调试模式下时，会输出一些和操作系统相关的信息，有时可能会和密码相关。因此，务必在上线时关闭调试模式。

②设置加密密钥，即 `SECRET_KEY`。你可能已经猜到了这个密钥会被用于存储用户密码。除此之外，它还会用于生成重置密码的 `token`、表单加密等。因此，在上线的时候，需要重新生成一个新的 `SECRET_KEY`，确保这个密钥没有在诸如 GitHub 上这样的开放网站。

另外，还有一个相当有意思的配置 `ALLOWED_HOSTS`，即只有在列表中的 `host` 才能访问，用于限定请求中的 `host` 值，以防止黑客构造包来发送请求。我们需要在这个配置里配置我们的域名，只允许使用访问这个应用的内容：

```
ALLOWED_HOSTS = [  
    '127.0.0.1',  
    'growth.studio.ren'  
]
```

从上面一个个修改配置的方法来看，理想的方式是创建一个额外的配置文件来存储这些内容，这也是我们在进入自动化配置以前需要做的。

6.1.3 配置管理

在上一节，我们仍然按小步前进的思想来展开内容。

①先使用 Nginx 部署一个 “hello, world”。

③在服务器上使用 Django WSGI 服务器来运行代码。

③使用 Gunicorn 作为 WSGI 服务器，并介绍其多线程能力来提高性能。

④使用 Circus 进行进程管理，来确保应用程序的进程。

在这个过程中会产生一堆配置文件，这些配置文件有相当多的内容是相互依赖的，如 Gunicorn 与 Nginx、Nginx 与 Web 应用的静态文件等。应用在开发过程中部署相关的配置

文件也会不断变化，这些变化也应该被记录下来。当线上的版本使用了新的配置，而我们没有对旧的配置进行备份，因出现严重 Bug 而导致下线时就难以即时恢复旧的环境。因此，我们仍然需要使用版本控制来管理这些配置文件。

这些脚本和配置本身也是项目代码的一部分，因此可以将这些配置及脚本直接加入项目，并用 Git 对它们进行管理。而当我们的代码里有一些密钥、Token 等内容时，就需要考虑一些额外的安全措施——我们不应该在公开的地方提交相关的代码，如 GitHub。同时，我们应该尽可能地生成额外的密钥配置文件，并保证这些配置只在需要的时候复制到服务器。

我们在第 4 章就介绍了如何在不同的环境里使用依赖管理文件 `requirements.txt`。这里还需要简单介绍一下如何在不同的环境里使用不同的配置文件。

要实现这一点有一个比较简单的做法，就是在代码库里为开发环境创建一个配置文件，而在一个额外的安全环境里创建额外的配置文件。随后，在 `settings.py` 文件里引入这个文件即可，代码如下：

```
try:
    from .local_settings import *
except ImportError as e:
    pass
```

如在本地的 `local_settings.py` 文件里可以将调试设为开，并使用 SQLite 作为数据库：

```
DEBUG = True

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
    }
}
```

在产品环境下，则设置好上文提到的 `SECRET_KEY`，并使用 MySQL 作为数据库等：

```
SECRET_KEY = ''

DEBUG = False

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': 'growth_studio',
        'USER': 'root',
        'PASSWORD': 'phodal',
        'HOST': '127.0.0.1',
        'PORT': '3306',
    }
}
```

这样既可保证本地运行环境的便利性，也使得应用在服务器上运行的时候是安全的。

6.2 自动化部署

实施自动化部署的过程中，我们需要了解一个很重要的概念，就是基础设施即代码。如 Martin Folwer 在《基础设施即代码》中所说：它是一种通过代码来定义计算和网络基础设施的方法，它可以应用于任何软件系统中。在这篇文章中提出了在自动化部署过程中，应该对部署代码考虑的实践：

- 使用描述文件，即配置信息应该存放于配置的描述文件中，如 shell 脚本等，这可以让我们自动化配置修改。
- 自文档化的系统和过程，即用代码来实现代码和部署过程。
- 版本化所有的代码。
- 持续地测试系统和过程。

- 小步前进。
- 保证服务持续可用。

在手动部署的过程中，我们已经将配置存档于配置文件中，还需要做一些改进来提高这个自动化的过程。

- 将配置文件存放在项目代码里。
- 使用 Fabric 来编写自动化脚本。
- 更新配置都有相应的任务。
- 每次修改配置应该做一次提交。
- 考虑在测试环境时，使用 UI 自动化测试来测试部署脚本。
- 对于大型的系统，采用蓝绿部署。

在自动化部署流程的时候，我们需要选择好一个自动化工具。这些自动化工具一般都会使用那些和命令行交互比较好的语言开发，诸如 Ruby 和 Python 语言等。在自动化运维方面，Python 语言使用频率相对比较高，在这方面的框架有：Ansible、Fabric、SaltStack。我们在项目中使用了 Fabric 来作为构建工具，因此将先使用 Fabric 来完成自动化部署。

6.2.1 使用 Fabric 自动化部署

对小型项目以及软件开发工程师来说，Fabric 是一个轻量级，并且容易上手的选择。我们只需要将部署转换为一个个步骤，再配合 Fabric 使用即可。下面回顾一下之前部署时的主要步骤。

- 安装应用运行的基础软件。
- 下载指定版本的 Web 应用包。

- 完成 Web 应用的初始化操作。

- 复制 Nginx、Circus 配置文件。

- 启动 Nginx 和 Circus。

在进入这些步骤之前，先让我们从一个简单的远程命令说起。

1. 一个简单的远程执行命令

在第 4 章介绍使用 Fabric 来运行脚本命令的时候，使用的是在本地环境下运行的 local API；当我们需要在远程服务器下执行类似的命令时，就需要使用 run 函数来执行。先让我们看一个简单的例子：

```
from fabric.api import run
```

```
@task
```

```
def host_type():
```

```
    run('uname -a')
```

这里的 **uname** 是在 Linux 和 UNIX 下用来获取电脑和操作系统相关信息的命令，**-a** 参数则表示详细输出所有的信息：内核名称、主机名、内核版本号、内核版本、硬件名、处理器类型、硬件平台类型、操作系统名称。

保存上述代码到 **fabfile.py** 文件里，然后就可以直接运行这个任务：

```
fab host_type -H 10.211.55.26 -u phodal
```

在上面的命令里，我们传了两个参数，使用 **-H** 传入了主机的 IP，使用 **-u** 参数传入用户名 **phodal**，也可以使用 **-p** 来传入密码。当执行上面的命令时，终端会返回下面的结果：

```
[10.211.55.26] Executing task 'host_type'
```

```
[10.211.55.26] run: uname -a
```

```
[10.211.55.26] Passphrase for private key:
```

```
[10.211.55.26] Login password for 'phodal':
```

```
[10.211.55.26] out: Linux ubuntu 4.8.0-22-generic #24-Ubuntu SMP Sat Oct 8
```

```
09:14:42 UTC 2016 i686 i686 i686 GNU/Linux  
[10.211.55.26] out:
```

在上面的日志里，我们可以看到每一行都是以主机 IP 开始的，以 out: 开始的内容则是服务器返回的文本。输入密码后，服务器就返回了主机的相关信息，如操作系统 Ubuntu、内核 4.8.0-22 等内容。由于在我的机器上为密钥设置了一个密码，因此与一般的步骤相比多了第三行 Passphrase for private key。

接着，让我们来看一个不安全的自动登录的例子，即请不要在代码中存储用户名和密码。在 fabfile 中使用全局的类字典对象 env 传入 hosts、user、password 等信息：

```
env.hosts = ['10.211.55.26']  
env.user = 'phodal'  
env.password = '940217'
```

上面的代码使用 hosts 来存储 IP 信息，你可以看到这是一个数组，因此可以传入多个 IP 信息。同理于用户名和密码，我们都可以使用数组的形式来传入。然后只需要再次执行 fab host_type 就可以。

接着，根据这个简单的 Demo 来完成更多的工作。我们将部署的过程拆分为两部分内容：

- 运行环境搭建。
- 编写自动化部署脚本。

这主要是由于环境搭建在一个服务器上只需要一次，而部署则会在一个服务器上执行多次。

2. 运行环境的搭建

在当前背景下，搭建运行环境只需要考虑以下 4 点。

- 安装相应的 Ubuntu 软件，如 git、python3-pip、nginx 等。
- 安装 virtualenv 软件，并使用该软件来创建虚拟环境。

- 安装 circus。
- 删除默认的 Nginx 配置。

因此，我们可以根据这几个步骤来编写 task：

```
nginx_enable_path = "/etc/nginx/sites-enabled/"

@task
def setup():
    """ Setup the Ubuntu Env """
    sudo('apt-get update')
    APT_GET_PACKAGES = [
        "build-essential",
        "git",
        "python3-dev",
        "python3-pip",
        "nginx",
        "virtualenv",
    ]
    sudo("apt-get install -y " + " ".join(APT_GET_PACKAGES))
    sudo('pip3 install circus')
    sudo('rm ' + nginx_enable_path + 'default')
    run('virtualenv --distribute -p /usr/bin/python3.5 py35env')
```

首先，用 `apt-get update` 更新 Ubuntu 的软件索引。接着，安装需要的 Ubuntu 软件，如 `git`、`nginx`、`virtualenv`、`python3-pip` 等，其中的 `build-essential` 是一系列工具集——在编译软件时需要用到的，`python3-dev` 则是在编译 Python 模块需要用到的。随后，用 `pip3` 安装 `circus`，并删除默认的 Nginx 配置，以便部署的时候更新它。最后创建一个基于 Python 3.5 版本下的虚拟环境。

3. 编写自动化部署脚本

我们可以依据基础设施即代码中提到的自文档化的系统和过程原则，用代码来表示

文档:

```

app_path = "~"

@task
def deploy(version):
    """ depoly app to cloud """
    with cd(app_path):
        get_app(version)
        setup_app(version)
        config_app()

        nginx_config()
        nginx_enable_site('growth-studio.conf')

        circus_config()
        circus_upstart_config()

        circus_start()
        nginx_restart()

```

上面的代码是我们在手动编写一个个部署的步骤后重构出来的。原先的代码则是根据部署 Web 应用的流程整合而成的:

```

@task
app_path = "~"

def deploy(version):
    """ depoly app to cloud """
    with cd(app_path):
        run(('wget ' + 'https://codeload.github.com/phodal/growth_studio/'
tar.gz/v' + '%s') % version)
        run('tar xvf v%s' % version)

```

```

with prefix('source ' + virtual_env_path):
    run('pip3 install -r growth-studio-%s/requirements/prod.txt' %
version)

    run('rm -f growth-studio')
    run('ln -s growth-studio-%s growth-studio' % version)

with cd('growth-studio'):
    run('python manage.py collectstatic -l --noinput')
    run('python manage.py migrate')

...

```

这里就不重复描述这部分内容了，在上面的代码中使用了 `with` 则 `prefix` 语句块：

- `with` 设置当前工作环境的上下文，如上面的 `cd(app_path)` 表示下面执行的每个脚本都在这个路径下执行。
- `prefix` 设置命令执行的前缀，则每次都在执行 `run` 函数之前，先执行一次激活虚拟环境的操作。

对 Nginx 配置来说，我们是将项目里的配置文件复制到远程服务器上，然后创建一个软件链接，代码如下：

```

nginx_config_path = os.path.realpath('deploy/nginx')
nginx_available_path = "/etc/nginx/sites-available/"
nginx_enable_path = "/etc/nginx/sites-enabled/"

def nginx_config(nginx_config_path=nginx_config_path):
    "Send nginx configuration"
    for file_name in os.listdir(nginx_config_path):
        put(os.path.join(nginx_config_path, file_name), nginx_available_
path, use_sudo=True)

```

```
def nginx_enable_site(nginx_config_file):
    "Enable nginx site"
    with cd(nginx_enable_path):
        sudo('rm -f ' + nginx_config_file)
        sudo('ln -s ' + nginx_available_path + nginx_config_file)
```

在初始化的时候，我们可以直接执行 `fab setup` 创建环境；执行 `fab deploy:x.x.x` 则用于部署指定版本的 Web 应用，如 `fab deploy:0.0.6`。

由于这部分内容和手动部署比较相似，读者可以直接从源码中获取这些内容，这里就不展开详细讨论。需要注意的是，由于笔者没有大规模部署的经验，因此，建议有这方面需要的读者可以阅读更专业的书籍。

6.2.2 探索更优雅的方案

我们在最开始的时候采用 Fabric，而不是诸如 Ansible 工具的原因是：从手动输入运维命令到全自动运维，我们需要有一个中间的过程，而 Fabric 则可以提供这个过渡。虽说 Fabric 是这个过程中的一个过渡，它仍然非常适合当前的情形。使用自动化配置工具 Ansible 则可以将部署的过程变成一个配置文件。

与此同时，我们在运行的时候局限于 Ubuntu 操作系统。当我们打算迁移到其他操作系统上时，就不得不修改这些自动化脚本——多数时候相当于重写。因此，我们可以考虑使用 Docker 来隔离操作系统。

1. 试用 Ansible

从上面的部署流程可以看到，我们所做的事情莫过于：安装软件、执行脚本、更新配置文件、重启服务等，这些实际上都可以看作是配置。而使用系统自动化工具 Ansible 则可以将这些都配置化，它和 Fabric 一样都是基于 Python paramiko 开发，都可以在本地的机器上直接运行。与编写部署过程大有不同的是：使用 Ansible 时，我们只需要编写一个名为 `playbook` 的领域特定语言配置即可。如下是安装 Nginx 的一个示例：

```
---
- name: Configure webserver with nginx
  hosts: webserver
  become: yes
  become_method: sudo
  tasks:
    - name: install nginx
      apt: name=nginx update_cache=yes
    - name: copy nginx config file
      copy: src=deploy/nginx/growth-studio.conf dest=/etc/nginx/sites-
available/default
    - name: enable configuration
      file: >
        dest=/etc/nginx/sites-enabled/default
        src=/etc/nginx/sites-available/default
        state=link
```

在这个配置文件里，一共做了三件事：安装 Nginx、复制配置文件、让配置生效。与上面我们写函数不同的是，这里的步骤都是使用配置文件来实现的。然而我们可以发现思想上是一样的，只是表达方式不同罢了。因此很容易将上面的代码转换成 Ansible 中的 playbook，这个任务交由读者来练习了。

编写完脚本后，配置 hosts 以及 Ansible 配置文件 ansible.cfg，再执行如下脚本：

```
ansible-playbook playbook.yml --ask-become-pass
```

就可以在服务器安装我们的应用了。

2. 结合 Docker 简介

如果你已经使用了 Ansible 来完成自动化部署，就可以考虑使用 Docker 来完成下一步操作。Docker 是一个基于 LXC 技术之上构建的 Container 容器引擎，简单地说，是一个基于操作系统的虚拟机，但是性能和在正常服务器上运行相差无几。使用 Docker 打包应用就相当于将操作系统与应用打包在一起。

Docker 使用了一个名为 Dockerfile 的文件来对镜像进行配置，其处理方式也与 Fabric 和 Ansible 很相似。如下是在 Ubuntu 镜像下安装 Nginx 的一个示例：

```
FROM dockerfile/ubuntu

RUN apt-get update && \
    apt-get install -y nginx

VOLUME ["/etc/nginx/sites-enabled", "/etc/nginx/certs", "/etc/nginx/conf.d",
"/var/log/nginx", "/var/www/html"]

WORKDIR /etc/nginx

CMD ["nginx"]

EXPOSE 80
EXPOSE 443
```

其本质仍然是通过编写类似于 Shell 的脚本来完成工作。在使用 Docker 部署时，我们可以直接将应用打包成镜像，打包完估计在 100MB 左右，或者是基于某个原有的镜像，在这个镜像上运行部署脚本。

由于 Docker 本身是一个比较复杂的话题，因此这里就不展开讨论了。不过我们建议读者不要直接使用 Dockerfile 来完成部署，而是结合 Fabric 或者 Ansible 来完成构建镜像。

6.3 隔离与运行环境

为了将我们的应用部署到服务器上，我们花费了大量的时间为其配置一个运行环境。理想情况下，我们应该使应用部署起来不会受环境影响，这时就要为应用创建好不同的几个运行环境。我们编写了 Fabric 脚本来隔离操作系统与 Web 应用，使用 VirtualEnv 来创建虚拟环境隔离不同应用的依赖包。在一些测试环境里，我们还会使用虚拟机来隔离环

境和计算机。从系统的底层到应用的顶层，我们会发现这其中开发人员使出浑身解数来保证运行环境的独立。

- ①隔离硬件：虚拟机。
- ②隔离操作系统：容器虚拟化。
- ③隔离底层：应用容器。
- ④隔离依赖版本：虚拟环境。
- ⑤隔离运行环境：语言虚拟机。
- ⑥隔离语言：DSL。

从一个大的流程来看，这实际上是一个请求的处理过程，一个 HTTP 请求会先到达你的主机。如果你的主机上运行多个虚拟机实例，请求就会来到这个虚拟机上；或者你是在 Docker 这一类容器里运行你的程序，请求也会先到达 Docker。随后这个请求就会交由 HTTP 服务器来处理，如 Apache、Nginx，这些 HTTP 服务器再将这些请求交由对应的应用或脚本来处理。随后将交由语言底层的指令来处理，如图 6-7 所示。

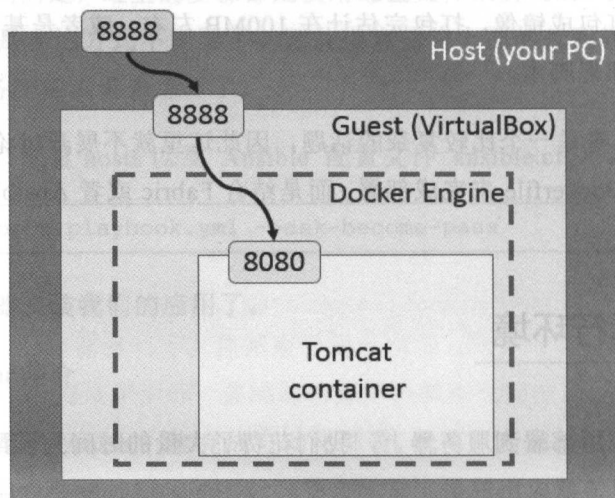


图 6-7 Docker Tomcat

不同的环境有不同的选择，当然也可以结合在一起。不过，从理论上说，在最外层还是应该有一个真机。下面详细介绍不同环境的隔离方案。

1. 隔离硬件：虚拟机

在虚拟机技术出现之前，为了运行不同用户的应用程序，人们需要不同的物理机才能实现这样的需求。对 Web 应用程序来说，有的用户的网站访问量少，消耗的系统资源也少，有的用户的网站访问量大，消耗的系统资源也多。虽然有不同类型的服务器可以选择，但对多数的访问量少的用户来说，他们需要支付同样的费用。这听上去很不合理，并且也浪费了大量的资源，对系统管理员来说，管理这些系统也不是一件容易的事。在过去硬件技术革新特别快，让操作系统运行在不同的机器上也不是一件容易的事。

虚拟机（Virtual Machine）指通过软件模拟的具有完整硬件系统功能的、运行在一个完全隔离环境中的完整计算机系统。

这是一个很有意思的技术，它可以让我们在一个主机上同时运行几个不同的操作系统。我们可以为这几个操作系统使用不同的硬件，在这之上的应用可以使用不同的技术栈来运行，并且从理论上互相不影响，其架构如图 6-8 所示。

借助虚拟机技术，当我们需要更多的资源时，创建一个新的虚拟机就行。同时，由于这些虚拟机上运行的是同样的操作系统，并且可以使用相同的配置，我们只需要编写一些脚本就可以实现其自动化。当物理机发生问题时，可以很快将虚拟机迁移或恢复到其他宿主机上。

2. 隔离操作系统：容器虚拟化

对大部分开发团队来说，直接开发基于虚拟机的自动化工具不是一件容易的事，并且它从使用成本上说比较高。这时就需要一些更轻量级的工具容器——它可以提供轻量级的虚拟化，以便隔离进程和资源，而且不需要提供指令解释机制和全虚拟化的其他复杂性。另外，它从启动速度上说也更快。



图 6-8 虚拟机

(1) LXC

在介绍 Docker 之前，我们还是稍微提一下 LXC。因为在过去我有一些使用 LXC 的经历，让我觉得 LXC 很赞。

LXC 的名称来自 Linux 软件容器（Linux Containers）的缩写，即一种操作系统层虚拟化（Operating system-level virtualization）技术，为 Linux 内核容器功能的一个用户空间接口。它将应用软件系统打包成一个软件容器（Container），内含应用软件本身的代码，以及所需要的操作系统核心和库。通过统一的名字空间和共用 API 来分配不同软件容器的可用硬件资源，创造出应用程序的独立沙箱运行环境，使得 Linux 用户可以很容易地创建和管理系统或应用容器。

我们可以将上面说到的虚拟机做一个简单对比，其架构如图 6-9 所示。

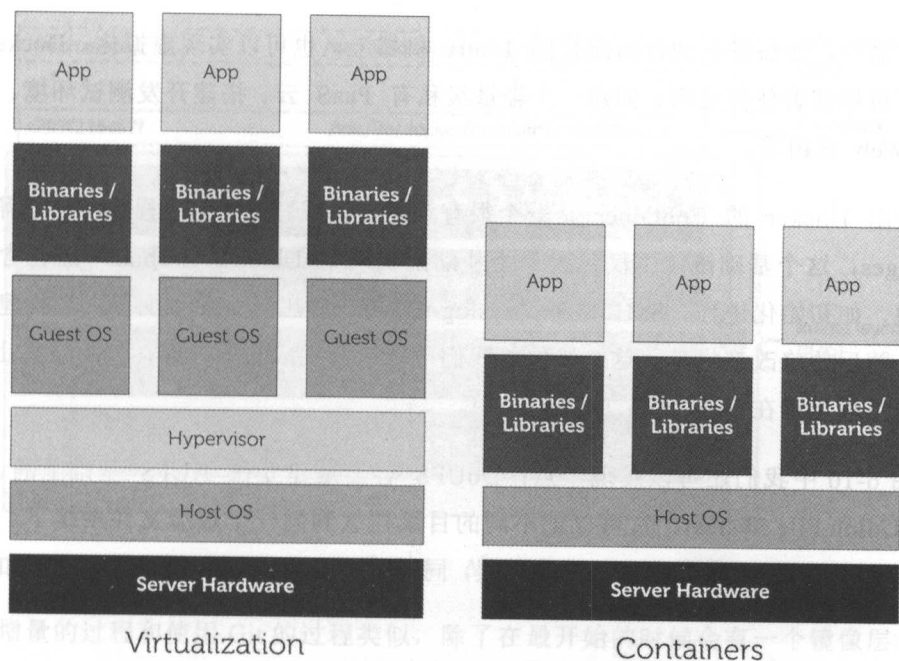


图 6-9 LXC vs VM

我们会发现虚拟机中多了一层 Hypervisor——运行在物理服务器和操作系统之间，它可以让多个操作系统和应用共享一套基础物理硬件。这一层级可以协调访问服务器上的所有物理设备和虚拟机，然而由于这一层级的存在，它也将消耗更多的能量。据爱立信研究院和阿尔托大学发表的论文表示：Docker、LXC 与 Xen、KVM 在完成相同的工作时要少消耗 10% 的能耗。

LXC 主要是利用 cgroups 与 namespace 的功能，来向提供应用软件一个独立的操作系统运行环境。cgroups（即 Control Groups）是 Linux 内核提供了一种限制、记录、隔离进程组所使用的物理资源的机制。由 namespace 来进行隔离控制。

与虚拟机相比，LXC 隔离性方面有所不足，这就意味着在实现可移植部署时会遇到一些困难。这时就需要 Docker 提供一个抽象层和一个管理机制。

(2) Docker

Docker 是一个开源的应用容器引擎，让开发者可以打包其应用以及依赖包到一个可

移植的容器中，然后发布到任何流行的 Linux 机器上，也可以实现虚拟化。Docker 可以自动化打包和部署任何应用、创建一个轻量级私有 PaaS 云、搭建开发测试环境、部署可扩展的 Web 应用等。

构建出 Docker 的 Container 是一个很有意思的过程。在这个过程中，首先需要有一个 base images，这个基础镜像不仅包含一个基础系统，如 Ubuntu、Debian，还包含了一系列的模块，如初始化进程、SSH 服务、syslog-ng 等一些工具。由上面原内容构建一个基础镜像，随后的修改都将基于这个镜像，我们可以用它生成新的镜像，一层层往上叠加。而用户的进程运行在 writeable 的 layer 中。

从图 6-10 中我们还可以发现一点：DoUFS 容器是建立在 AUFS 基础上的。AUFS 是一种 Union File System，它可以把不同的目录挂载到同一个虚拟文件系统下。它的目的就是为了实现图 6-10 中增量递增的过程，同时又不会影响原有的目录，流程如图 6-11 所示。

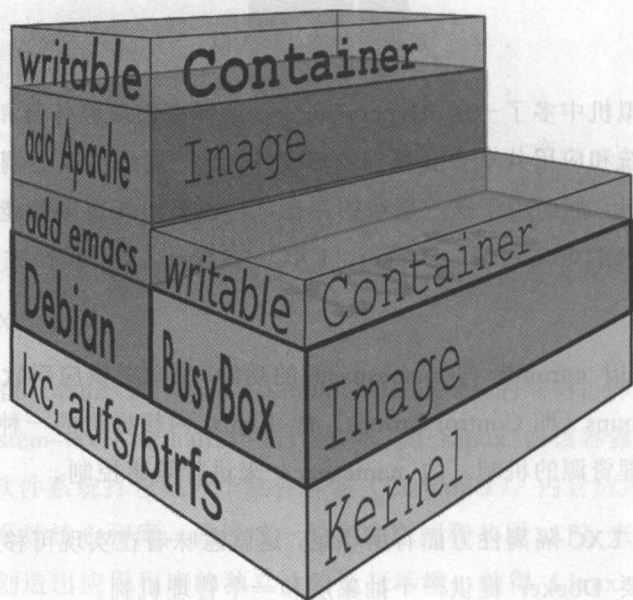


图 6-10 Docker Container

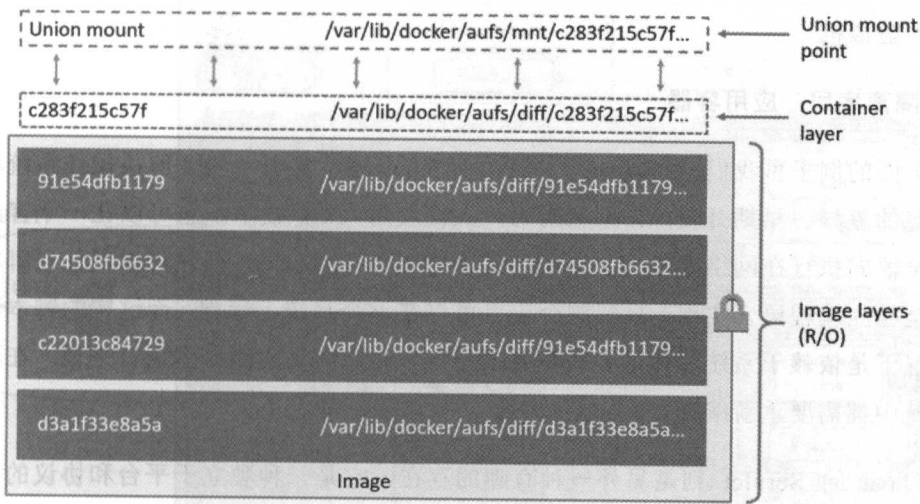


图 6-11 AUFS 层

其增量的过程和使用 Git 的过程类似，除了在最开始的时候会有一个镜像层。随后我们的修改都可以保存下来，并且当再次提交修改的时候，可以在旧有的提交上运行。

因此，Docker 与 LXC 的差别如图 6-12 所示。

Key differences between LXC and Docker

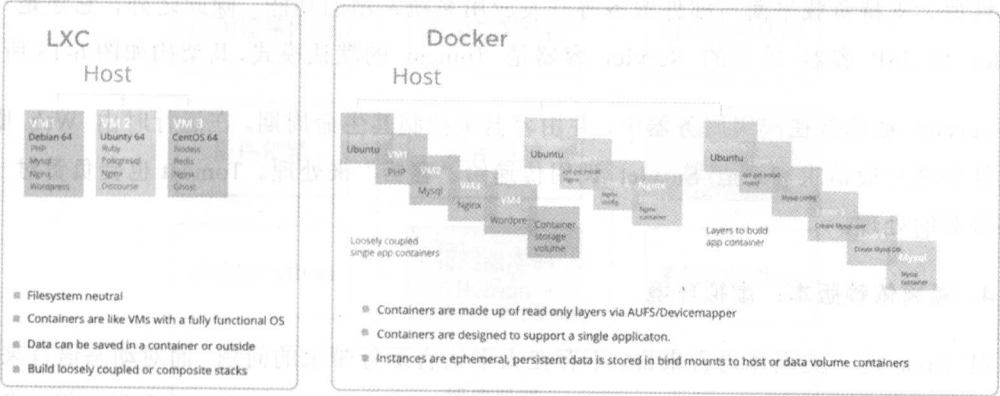


图 6-12 LXC 与 Docker

LXC 中的单个虚拟机只能是一个独立的虚拟机，而在 Docker 中则是一个时间线上

的一系列虚拟机。

3. 隔离底层：应用容器

在上面的例子里我们已经隔离开了操作系统的因素，接着还需要解决操作系统和开发环境引起的差异。早期开发 Web 应用时，人们使用 CGI 技术，它可以让一个客户端从网页浏览器向执行在网络服务器上的程序请求数据，并且 CGI 程序可以用任何脚本语言或者完全独立编程语言实现，只要这个语言可以在这个系统上运行。而这样的脚本语言在多数情况下是依赖于系统环境的，特别是对于 C++ 这一类的编译型语言来说，在不同的操作系统中都需要重新编译。

而 Java 的 Servlet 则是另外一种有趣的存在，它是一种独立于平台和协议的服务器端的 Java 应用程序，可以生成动态的 Web 页面。

在开发 Java Web 应用的过程中，我们在开发环境使用 Jetty 来运行服务，而在生产环境使用 Tomcat 来运行，它都是 Servlet 容器，可以运行同一个 Servlet 应用。Servlet 是指由 Java 编写的服务器端程序，它们是为响应 Web 应用程序上下文中的 HTTP 请求而设计的。它是应用服务器中位于组件和平台之间的接口集合。

Tomcat 服务器是一个免费的开源 Web 应用服务器。它运行时占用的系统资源小，扩展性好，支持负载平衡与邮件服务等开发应用系统常用的功能。除此之外，它还是一个 Servlet 和 JSP 容器，独立的 Servlet 容器是 Tomcat 的默认模式，其架构如图 6-13 所示。

Servlet 被部署在应用服务器中，并由容器来控制其生命周期。在运行时由 Web 服务器软件处理一般请求，并把 Servlet 调用传递给“容器”来处理。Tomcat 也会负责对一些静态资源的处理。

4. 隔离依赖版本：虚拟环境

对 Java 这一类编译语言来说，不存在太多语言运行带来的问题。而对动态语言来说，就存在这样的问题，如 Ruby、Python、Node.js 等，这个问题主要集中于开发环境。当然，如果你在一个服务器上运行几个不同的应用，也会存在这样的问题。这一类工具在 Python 里有 VirtualEnv，在 Ruby 里有 RVM、rbenv，在 Node.js 里有 NVM。

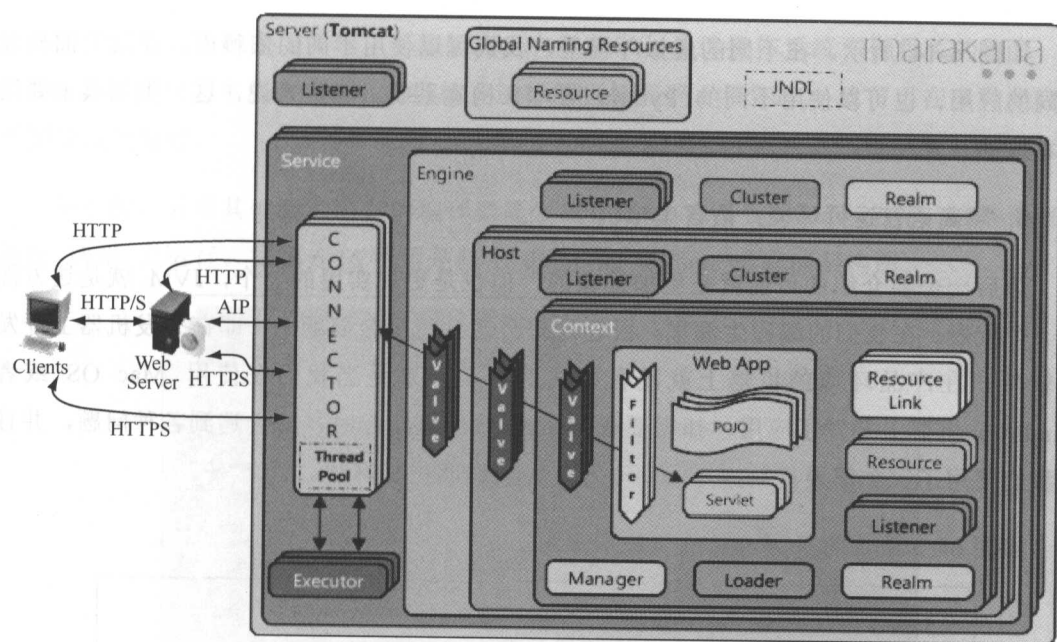


图 6-13 Tomcat 架构

图 6-14 是使用 VirtualEnv 时几个不同应用的架构图。

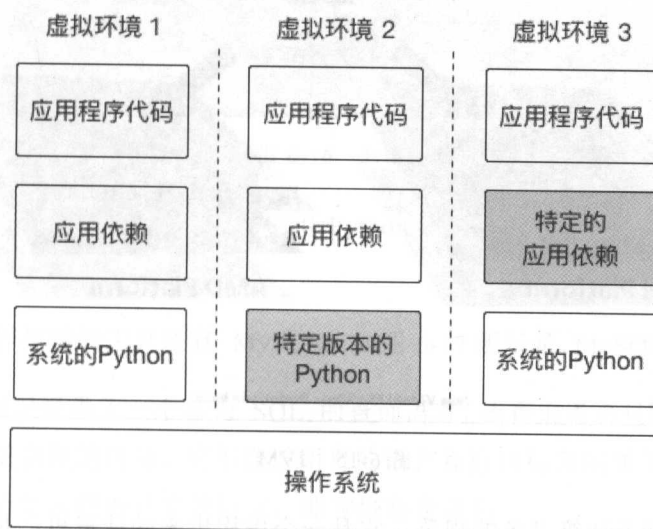


图 6-14 VirtualEnv

如图 6-14 所示，在不同的虚拟环境里，我们可以使用不同的依赖库。在这上面构建不同的应用，也可以使用不同的 Python 版本来构建系统。通常来说，这一类工具主要用于本地的开发环境。

5. 隔离语言运行环境：语言虚拟机

最后一个要介绍的可能就是更加抽象的，但也是更加实用的一个，JVM 就是这方面的一个代表。在我们的编程生涯里，很容易就会遇到跨平台问题——即在开发机器上开发的软件，在产品环境的机器上就没有办法运行。特别是当我们在使用 Mac OS 或者 Windows 机器上开发了应用，却需要在 Linux 系统上运行时，就会遇到各种问题，并且当我们使用了一个需要重新编译的库时，这种问题就更加麻烦。

如图 6-15 所示的是 JVM 的架构示意图。

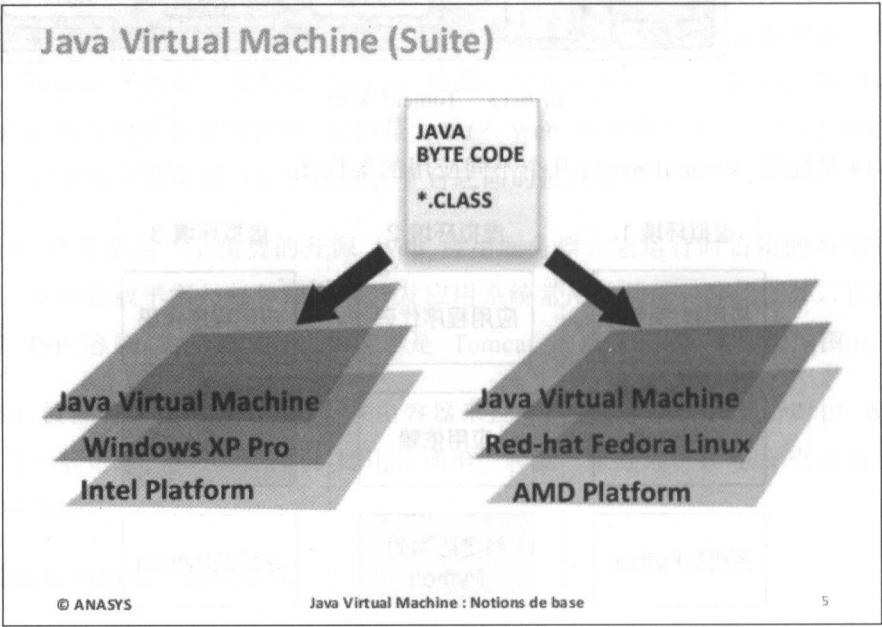


图 6-15 JVM

JVM 是一种用于计算设备的规范，它是一个虚构出来的计算机，是通过在实际的计算机上仿真模拟各种计算机功能来实现的。它可以实现“编写一次，到处运行”。

换句话说，它在底层实现了环境隔离，它屏蔽了与具体操作系统平台相关的信息，使得 Java 程序只需生成在 Java 虚拟机上运行的目标代码（字节码），就可以在多种平台上不加修改地运行。

基于此，只要其他编程语言的编译器能生成正确的 Java bytecode 文件，这种语言也能在 JVM 上运行。如图 6-16 所示是基于 JVM 的 Jython 语言的架构图。

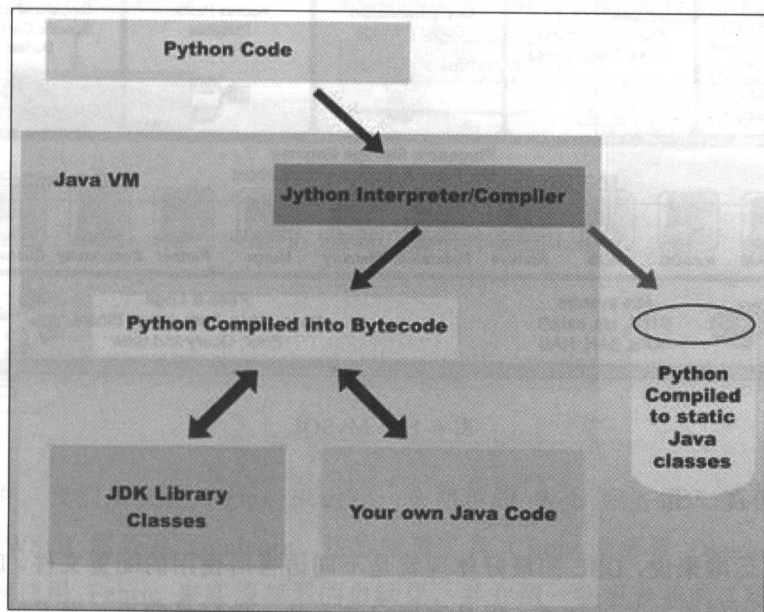


图 6-16 Jython

其底层是基于 JVM，而编写时则使用 Python 语言，并且它可以使用 Java 的模块来编程。

常见拥有同样架构的工具还有 MySQL，如图 6-17 所示是 MySQL 的架构图。

MySQL 在顶层提供了一个名为 SQL 的查询语言，该查询语言只能用于查询数据库，然而它却是一种更高级的用法。它不像通用目的语言那样目标范围涵盖一切软件问题，而是专门针对某一特定问题的计算机语言，即领域特定语言。

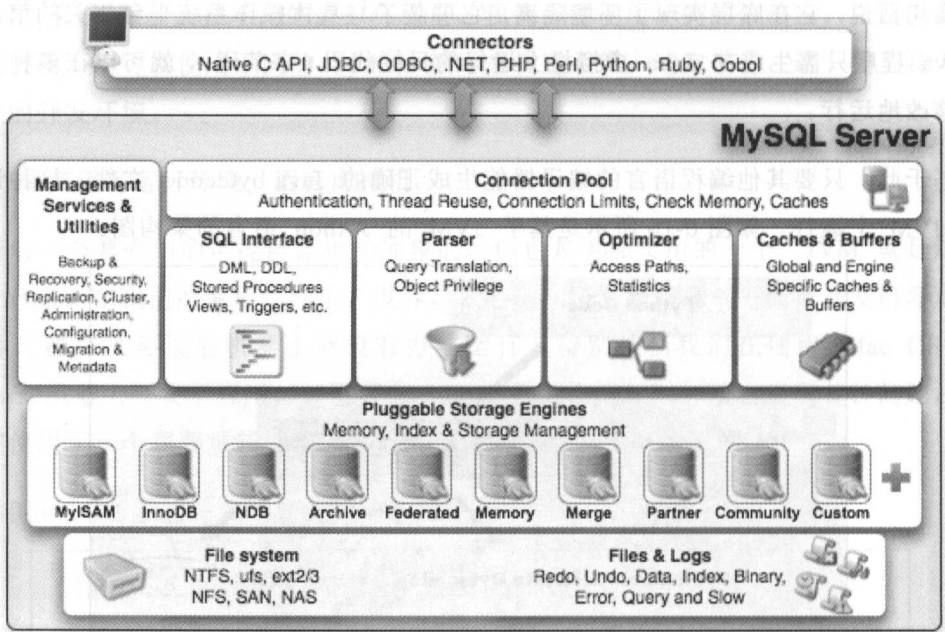


图 6-17 MySQL

6. 隔离语言：SL

对自动化运维来说，DSL 的最好体现就是不同语言所使用的配置文件。DSL 是一项特别有意思也特别值得期待的技术，但是实现它并不是一件容易的事。为了讨论隔离环境的一部分，我们只看外部 DSL。内部 DSL 与外部 DSL 最大的区别在于：外部 DSL 近似于创建了一种新的语法和语义的全新语言。如图 6-18 所示是两种 DSL 的对比。

在这样的外部 DSL 里，我们有自己的语法、自己的解析器、类型检测器等。最简单且最常用的 DSL 就是 Markdown，我们可以将它直接转化 HTML、PDF 或者其他格式的文本。如果我们可以将业务逻辑写成 DSL，就不需要担心底层语言的变动过多会影响原有的业务逻辑。换句话说，这相当于创建了自己的语言隔离环境，我们不需要思考用何种语言来使用我们的业务。

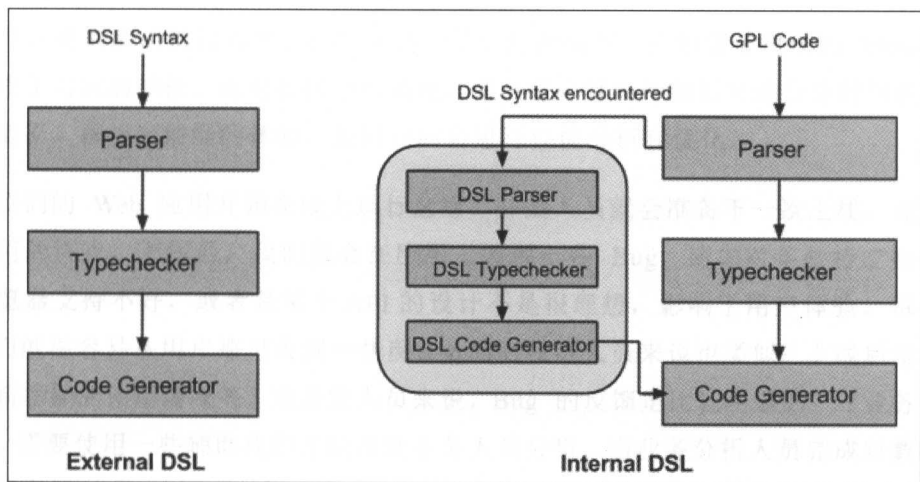


图 6-18 内部 DSL 和外部 DSL

6.4 小结

在本章中，我们从安装 Nginx 来运行一个简单的 Web 服务器入手；随后详细介绍了如何结合 WSGI 服务器 Gunicorn、进程管理工具 Circus 来部署 Django 应用；同时，还介绍了如何使用 Fabric 来实现部署的自动化，并介绍一些更专业的运维工具来完成自动化。最后，对自动化部署做了一个总结，即自动化部署实际上是完成应用与环境的隔离。由于本章内容比较烦琐，并且容易出错，建议读者可以先在虚拟机上运行学习。

推荐阅读

《鸟哥的 Linux 私房菜》：这是一本很受欢迎的 Linux 运维基础入门书籍。

《奔跑吧 Ansible》：介绍了如何用 Django 和 Ansible 来完成自动化运维，以及如何与 Docker 及 Amazon EC2 等结合使用。



第 7 章

数据分析和性能优化

本章将先用 Google Analytics 为我们的应用添加数据分析服务，介绍如何用 Google Analytics 来分析网站数据、用户行为等数据；接着介绍开源分析平台 Piwik，为不能使用 Google Analytics 服务的用户提供一个解决方案；还将引入性能分析框架，如 OneAPM 或者 New Relic 来分析应用中的瓶颈，以帮助开发者优化程序，介绍一些常见的缓存策略来加速应用。

在开发博客应用的过程中，我们先创建了后台的模型，再创建了对应的 View 函数，最后创建了对应的模板。而对数据分析来说，则希望可以由外到里来进行分析与优化——先分析数据，再提高前端的体验，最后对后台进行性能分析和优化。

当我们的 Web 应用开始在线上运行之后，开发人员就会准备下一次上线。而上线并不是应用的终点，上线后，我们会在应用上发现一些 Bug，诸如对某些特定操作系统上的浏览器支持不好，或者是某个 API 的设计不是很理想，影响了用户体验。每次上线后，我们就很容易从用户那里收到一些反馈。而对业务人员来说也类似，上线后就要分析用户对新功能的使用情况等。对开发人员来说，Bug 的反馈是比较明显的；对业务分析人员来说，需要使用一些辅助代码才能帮助业务人员分析。当业务分析人员完成对数据的分析后，就会提出一些新的需求，再由开发人员来实现这个需求，最后形成如图 7-1 所示的环路。



图 7-1 精益环路

图 7-1 中数据部分占据了 1/3 的内容，贯穿了对产品的数据收集，以及到设计出新的功能。我们可以将这个步骤直接划分到数据分析，而要完成整个数据分析的过程，需要以下步骤。

- 识别需求，即我们所要分析的内容是什么。
- 收集数据，使用一些工具收集用户数据及使用情况等信息。
- 存储数据。
- 分析数据，从大量的数据中筛选中需要的部分数据。
- 展示数据，借助数据可视化工具来识别出重要的点。

在这个过程中将会涉及方方面面的知识，由于数据分析会涉及统计、编程、数据可视化等领域。因此，在本章将数据分析进行简要介绍，并介绍一些实用的工具来帮助我们进行数据分析，内容如下。

- 使用 Google Analytics / Piwik 对网站流量进行统计。
- 使用自定义的事件来追踪一些关键用户行为。
- 分析网站的流量来源、受众概览、转化率。

与此同时，在本章也将简要介绍一些可以提高网页性能及性能的方法：

- 介绍如何使用 PageSpeed 进行衡量和优化。
- 了解前端常用的一些优化技巧。
- 介绍并使用 APM 来对应用的瓶颈进行分析、优化。
- 介绍如何应用缓存来优化应用的性能。

除此之外，我们还将针对那些需要搜索引擎优先（SEO）的网站，提供一个快速的 SEO 入门。

7.1 网站监测与分析

从业务的角度来说，如果知道用户为什么在某一些页面上停留的时间比较长，我们就可以通过它分析为什么用户会停留这么久，并做出更好的优化方案。从优化体验的角度来说，如果知道用户经常在注册的过程中放弃了注册，那么说明这个注册的表单是有问题的，需要重新设计。

除此之外，当我们与一些机构合作进行网站、应用推广时，需要知道这些效果带来了多少用户。这些用户又有多少人真正使用了我们的产品，从相应的事件、动作与访问量进行对比，就可以知道网站的转化率是多少。

在这些网站监测工具里，Google Analytics 是比较流行的一个，它是由 Google 所提供的网站流量统计服务。尽管 Google Analytics 功能比较强大，但是受限于网络原因，以及一些公司内部制定的安全策略，我们使用不了这个服务。因此，我们将介绍一个开源的解决方案 Piwik，这样就可以在自己的服务器上存储用户数据，并保存服务的可用性。

如果我们只想简单地统计页面的访问流量，那么上述方案对我们而言就有些大材小用。我们可以创建一张简单的图片，同时在后台计算出相关图片的访问量，就可以测出网站的访问量。

7.1.1 Google Analytics

我第一次使用 Google Analytics 时，是因为我的博客需要一个访问量统计工具。通过这个工具，我就可以知道每天有多少人访问我的网站。随后我发现，它不仅可以查看多少人访问、页面的访问量、用户所在地区等基本内容，还可以知道用户的流量来源——是从搜索引擎过来的，还是从某个社交网站等；用户在这个网站停留了多久、网站的访问速度等。

在工作之后，我发现它还可以测量用户对某一类特定事件的追踪，如点击登录时可以向服务器发送一个登录事件。通过这个功能，我们可以很容易测出有多少人登录了网站，在登录的过程中又有多少人的验证码错误了等详细信息。

1. 添加 Google Analytics 服务

在使用 Google Analytics 之前，需要注册一个 Google 账号，这里假定读者可以访问 Google，并且已经有 Google 账号。我们只需要访问 <https://www.google.com/analytics/>，并通过简单的几步就可以注册使用 Google Analytics 的服务：

- 进入管理页面。
- 创建新账号。
- 填写账户名称等相关信息。
- 获取跟踪 ID。

界面如图 7-2 所示。



图 7-2 Google Analytics 获取跟踪 ID

可以获得如下脚本:

```
<script>

(function(i,s,o,g,r,a,m){i['GoogleAnalyticsObject']=r;i[r]=i[r]||function(){
  (i[r].q=i[r].q||[]).push(arguments)},i[r].l=1*new
  Date();a=s.createElement(o),

m=s.getElementsByTagName(o)[0];a.async=1;a.src=g;m.parentNode.insertBefore(a,m)
  })(window,document,'script','https://www.google-analytics.com/analytics.js','ga');

  ga('create','UA-87762194-1','auto');
  ga('send','pageview');

</script>
```

将这个脚本添加到模板文件里,就可以使用 Google Analytics 服务了。上面的代码可以分为两部分,第一部分从远程服务器获取 analytics.js 脚本,第二行 ga('create','UA-87762194-1','auto');设置了我的 Google Analytics 账号,第三行的 ga('send','pageview');则是向 Google Analytics 的服务器发送页面的 pageview,即用户访问页面的相关信息。

在添加完代码并上线几分钟后,就可以在 Google Analytics 的后台看到实时数据,如图 7-3 所示。

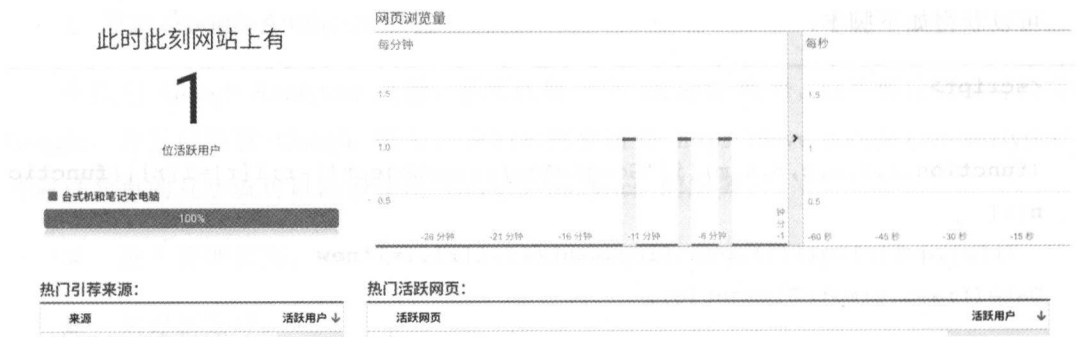


图 7-3 Google Analytics 实时数据

让我们看看为什么后台会接受到这些信息？我们只需要在网站上用鼠标右键单击“检查”，就可以显示出 Chrome 浏览器的调试端口。除了可以在这里调试应用，我们还可以查看网站发出的请求，可以看到一条发现 Google Analytics 的 URL 请求，内容如下：

```
https://www.google-analytics.com/r/collect?v=1&_v=j47&a=2088911285&t=pag
eview&_s=1&dl=http%3A%2F%2Fstudio.growth.ren%2F&ul=zh-cn&de=UTF-8&dt=Growth%
20Studio%20-%20Enjoy%20Create%20%26%20Share&sd=24-bit&sr=1440x900&vp=1316x32
7&je=0&fl=23.0%20r0&_u=AACAAMABI~&jid=574011281&cid=22433767.1470323209&tid=
UA-87762194-1&_r=1&z=409034613
```

从上面 URL 的参数里可以看到诸如：

- 当前网页的链接：http://studio.growth.ren/
- 所使用的语言：zh-cn
- 网站的编码格式：UTF-8
- 网站标题：Growth Studio - Enjoy Create & Share
- 屏幕分辨率：1440×900
- 显示器色彩：24-bit

等信息。除此之外，当我们访问这个网站的时候，服务器还会接收到一些更详细的信息，

如下是使用 Nginx 记录用户访问信息的一个日记：

```
183.39.153.133 - - [22/Nov/2016:14:24:18 +0000] "GET / HTTP/1.1" 200 4230
"- "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_6) AppleWebKit/537.36 (KHTML,
like Gecko) Chrome/54.0.2840.98 Safari/537.36" -
```

由前往后分别有用户的 IP、访问时间、HTTP 详细信息、浏览器详细信息等的内容。

从这些信息里可以分析出网站用户的一些信息：

- 用户所使用的浏览器、操作系统。
- 从访问的 IP 来获取用户的大致位置。
- 用户一般会在哪个时间访问网站。

除此之外，还可以知道哪些用户是多次访问等信息，而通过后台的分析工具可以知道更详细的内容。

2. 使用 Google Analytics 分析

为了展示如何从数据中获取一些有用的结论，需要有更多的数据作为支撑，因此，我将使用我的博客数据作为展示示例。在 Google Analytics 里有以下几个基本的分析大类用于分析数据：

- 实时，即展示当前正在访问网站的访客的信息：引擎来源、访问的网页、当前用户数、地点等。
- 受众群体，可以提供某一个时间段内的访客信息，并且可以使用不同维度的数据，（如会话、浏览量、时长等）来进行对比。
- 流量获取，展示用户是从哪里来到这个网站的，这部分信息对我们来说非常有帮助。
- 行为，即用户进行的一些操作，可以创建事件来追踪，一般会显示网页标题、链接等简单的信息。

- 转化，我们可以设定一些指标，用于衡量销售量、下载次数、视频播放次数等有价值的操作。

这些不同的类别对于不同角色的人来说，其用处是不一样的。

- 对开发人员来说，他们更关心用户使用的设备、操作系统和技术。
- 对业务人员来说，他们更关心用户的来源、转化率等信息。

下面将介绍几个基本的数据来展示如何使用这些数据进行分析。

(1) 受众群体

如图 7-4 所示是我的博客在过去一个月里“受众群体”概览。

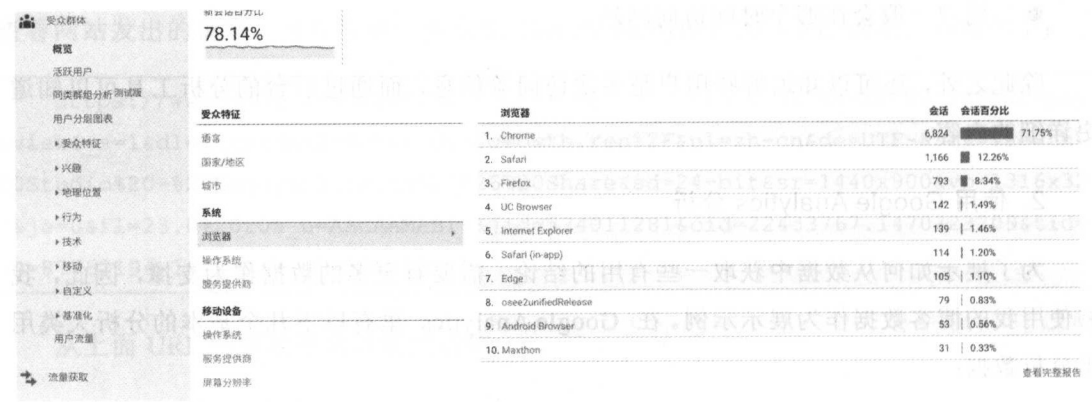


图 7-4 浏览器信息

从图 7-4 中可以看到用户使用的浏览器、操作系统、来源等信息，而作为一个开发人员，我们更关心用户所使用浏览器是什么？因此，从图 7-4 中整理出浏览器的使用情况，如表 7-1 所示。

从表 7-1 中可以发现，用户主要使用的是 Chrome 浏览器，有多达 71.75% 的用户，随后是 Safari、Firefox 浏览器。如果你对 Web 开发有一定了解，就知道 IE 在过去有着巨大的市场份额，但是在这个浏览器上的开发体验相当不好。过去版本的 IE（现在指的是 6~9）存在一系列的兼容性问题，这些兼容性问题是其不支持一些相关的标准，并且开发进

程滞后于其他浏览器所导致的。就我的技术博客而言，这些用户大多数都是开发人员，多数人都不会选择使用 IE。

表 7-1

浏览器	会话	会话百分比
Chrome	6,824	71.75%
Safari	1166	12.26%
Firefox	793	8.34%
UC Browser	142	1.49%
Internet Explorer	139	1.46%

因此，我们就可以有足够的理由来说服团队里的其他人：只提供有限的 IE 浏览器技术支持。对开发人员来说，使用诸如 Chrome、Firefox 这些现代的浏览器，我们可以使用大量的新特性来加速开发过程，并提供更好的用户体验。当然，如果你的数据结果表明有大量的 IE 用户，就需要在上面多花工夫。

(2) 流量获取

同样，我们可以从“流量获取”来分析用户的流量来源，如图 7-5 所示。

Default Channel Grouping	流量获取			行为			转化		
	会话	新会话百分比	新用户	跳出率	每次会话浏览页数	平均会话时长	目标转化率	目标达成次数	目标价值
	25,997 占总数的百分比: 100.00% (25,997)	79.74% 平均浏览次数: 79.70% (0.05%)	20,730 占总数的百分比: 100.05% (20,719)	74.17% 平均浏览次数: 74.17% (0.00%)	1.73 平均浏览次数: 1.73 (0.00%)	00:01:35 平均浏览次数: 00:01:35 (0.00%)	0.00% 占总数的百分比: 0.00% (0.00%)	0 占总数的百分比: 0.00% (US\$0.00)	US\$0.00
1. Organic Search	10,169 (39.12%)	80.18%	8,154 (39.33%)	81.36%	1.46	00:01:11	0.00%	0 (0.00%)	US\$0.00 (0.00%)
2. Direct	8,413 (32.36%)	83.22%	7,001 (33.77%)	73.96%	1.79	00:01:27	0.00%	0 (0.00%)	US\$0.00 (0.00%)
3. Referral	7,230 (27.81%)	75.31%	5,445 (26.27%)	64.23%	2.04	00:02:18	0.00%	0 (0.00%)	US\$0.00 (0.00%)
4. (Other)	115 (0.44%)	72.17%	83 (0.40%)	77.39%	1.46	00:01:31	0.00%	0 (0.00%)	US\$0.00 (0.00%)
5. Social	70 (0.27%)	67.14%	47 (0.23%)	74.29%	2.44	00:03:05	0.00%	0 (0.00%)	US\$0.00 (0.00%)

图 7-5 流量来源

图 7-5 是三个月内的博客流量数据，可以发现：

- 40%的流量来自于搜索引擎。

- 32% 的流量是直接访问的。
- 28% 的流量是从其他网站访问的。
- 社交网站的流量很少。

来自搜索引擎、直接访问、网站跳转的流量相差都不是特别多，然而这几种不同的流量是有所区别的。

- 搜索引擎：都是由用户在搜索引擎上搜索到网站的网页，并单击进入网站。我们只需要在搜索引擎的搜索结果页上拥有一个好的位置，就可以不断地获取流量，并且不需要额外的成本。也因此，要在搜索引擎上排得一个好的位置的难度也比较大。
- 直接访问：这些用户一般都是知道这个网站的人，或者是他们屏蔽了 Google Analytics 这一类工具。获取这一类用户的成本都比较高，这意味着用户已经知道这个网站，也知道这个网站对其的价值。
- 由其他网站访问而来：通常来说，这些用户通过一些外链来访问我们的网站，这些外链可能是我们和某一些网站合作引流过来的。这一类用户的获取难度就更大了，我们需要不断地在线上去推广网站。
- 社交网站：要在社交网站上获取流量，就意味着我们的社交账号（诸如微博）有相当多的粉丝，并且这些粉丝都对推广的内容有兴趣，才会通过链接访问我们的网站。除此之外，一些用户在社交网站上对文章的评论、转发等也可以增加流量。

因此，我们可以根据需要做不同的优化。对技术博客的主要流量来源就是用户在搜索引擎上搜索的结果，对有知名度的博客来说，可以通过 RSS 或者用户已经记住了网站；对诸如京东、淘宝这一类封闭的网站来说，他们的用户都是直接访问的，对搜索引擎进行优化不一定能取得太大的效果；而对淘宝网的专家来说，则需要通过社区网站、外部网站等让更多的用户访问他们的产品。

当然，对依赖搜索引擎进行导流的网站来说，也可以选择付费搜索来获得流量。

(3) 事件

先前我的博客并没有针对行为的一些监控，因为我们将使用“Growth：带你成为顶尖开发者”应用的数据来做一个简单介绍。要使用 Google Analytics 创建事件是很容易的，我们只需要在做相应操作的时候写如下一行代码即可：

```
ga('send', {
  hitType: 'event',
  eventCategory: 'User',
  eventAction: 'login',
  eventLabel: 'User Action'
});
```

这样我们就向服务器端发送了一个事件，这个事件的分类是 User，动作是 login，这样就可以记录用户的登录事件了。通过创建一系列的事件，可以获取相当多的用户操作的数据。

如图 7-6 所示是 Growth 应用在过去一个月里的事件数据。



图 7-6 事件统计

在这个 App 里，事件主要有六个类别，图 7-6 中的 Section、Discover Page、Community、User Center 和 Discover Ebook 五个类别，分别代表 App 的不同页面。从数据可以很容易知道：用户使用最多的就是 Section 和 Discover Page。除此之外，还有一个类别是用户的登录事件。然而这个事件触发得越少，就意味着用户登录越少。

当我们创建一个更细粒度的数据时，就可以分析出最受用户喜欢的功能。再结合用户体验设计师做一些用户调研，就能更容易地把握好产品的方向。

7.1.2 自建监测和分析服务

由于 Google 的服务在国内不可用，我们需要考虑其他的监测方案。除此之外，我们也更希望将用户的相关隐私数据存储在在自己的服务器上。这时，可以考虑使用 Piwik 搭建一个自己的服务。

当我们没有使用监测服务，但需要分析用户使用情况时，就可以考虑基于日志来对用户的信息进行简单分析。这时需要对日志进行分析，而通常来说，这些日志的文件都特别大，需要借助开源的日志管理方案 Elasticsearch + Logstash + Kibana，或者是 Spark、Flume 这样的工具来编写实时的日志分析。而在实现这些工具时，比较麻烦的就是进行数据可视化，如使用地图来显示不同地方的用户等。建议读者在具有一定的编程经验后再尝试，这样可以考查读者的全栈思维能力。

下面介绍基于 Piwik 自建监测服务。

Piwik 是一个开源的网站访问统计系统，基于 PHP 和 MySQL。与 Google Analytics 相似，它可以提供详细的统计信息，也可以提供搜索引擎关键词等流量分析功能。

与 Google Analytics 有很大区别的是：Piwik 支持插件扩展，并采用开放 API 架构，开发人员可以依照自己的需要来创建功能。这里跳过这部分安装过程，感兴趣的读者可以从本书附录 B 中获取安装 Piwik 的方法。

Piwik 也是采用在 HTML 中参加分析脚本的方法来添加分析服务，其代码如下：

```
<!-- Piwik -->
<script type="text/javascript">
  var _paq = _paq || [];
  _paq.push(['trackPageView']);
  _paq.push(['enableLinkTracking']);
```

```

(function() {
  var u="//10.211.55.26/";
  _paq.push(['setTrackerUrl', u+'piwik.php']);
  _paq.push(['setSiteId', '1']);
  var d=document, g=d.createElement('script'), s=d.getElementsByTagName
('script')[0];
  g.type='text/javascript'; g.async=true; g.defer=true; g.src=u+'piwik.js';
  s.parentNode.insertBefore(g,s);
})();
</script>
<noscript><p></p></noscript>
<!-- End Piwik Code -->

```

上面的代码和之前的 Google Analytics 的脚本差不多，我们将从 Piwik 服务器下载并分析脚本，然后执行这个脚本。因此，如果你想使用这样的分析服务，请确保 Piwik 服务器的访问速度够快。

与 Google Analytics 稍有区别的是，Piwik 也可以对那些不支持 JavaScript 或者早期的上网设备进行数据统计，即通过 `noscript` 方法：

```

<noscript><p></p></noscript>

```

当页面不支持 JavaScript 时，就会在网页内插入这个图片，这个图片的地址指向的是 Piwik 服务器。当浏览器访问我们的服务器时，就可以获取这个用户的一些相关信息。

图 7-7 是 Piwik 的后台，从这个后台可以看到实时的用户信息、过去时间段的用户访问、受众概览、受众地图等信息。由于这些数据与我们在 Google Analytics 后台看到的类似，这里就不展开详细介绍了。



图 7-7 Piwik 仪表盘

7.2 性能分析及优化

在 7.1 节中，我们使用的分析工具可以帮助改进产品。尽管它们也能告诉我们应用需要优化，但是无法告诉我们可以针对哪些细节进行优化。因此，本节将介绍如何对前端及后端的代码进行分析和优化。

- PageSpeed 可以帮助改善网页速度。
- New Relic 可以帮助分析应用中的瓶颈，让我们针对性地进行优化。

这两个工具并不局限于使用的语言和技术，我们可以很容易地将它们应用到现有的项目里。除 PageSpeed 以外，Yahoo 的 YSlow 也是一个非常不错的方案。

7.2.1 前端优化：用 PageSpeed 工具分析和优化

PageSpeed 是由 Google 推出的网页速度优化工具，它主要是基于首屏加载时间和完整的网页加载时间来对网页性能提升做出建议，包含服务器配置、网页的 HTML 结构及其所用的外部资源等。

Google 提供了几个不同的 PageSpeed Insights 分析工具，即网页版和插件版。网页版可以通过访问：<https://developers.google.com/speed/pagespeed/insights/?hl=zh-CN> 来进行测试，插件版可以直接在 Chrome 应用商店中安装。由于 PageSpeed 的插件更容易使用，并且不会受限于网络的影响，因此，推荐读者安装这个插件作为日常分析使用。

另外，PageSpeed 还针对 Apache 和 Nginx 服务器提供了自动化的优化模块，通过重新编译 HTTP 服务器，可以让它来帮助我们自动优化网页的速度。

1. 使用 PageSpeed Insights 进行分析

PageSpeed 使用分数来衡量网页的性能，分数则主要取决于下面两个时间。

- 首屏加载时间：从用户请求新页面到浏览器呈现首屏内容所用的时间。
- 完整的网页加载时间：从用户请求新网页到浏览器完全呈现网页所用的时间。

需要注意的是，由于网络性能存在较大的差异，这个工具并不会关注网络连接问题。因此，PageSpeed 在其得分说明中指出：

Page Speed 得分表示网页加载速度还能提升多少。高分表示提升余地不大，而低分表示提升余地较大。Page Speed 得分衡量的并不是网页的加载时间。

安装 PageSpeed 插件后，我们只需要在相应的网页里打开 Chrome 开发者工具，再找到对应的 PageSpeed 标签单点击插件中的“开始分析”按钮，就可以对网页进行分析。图 7-8 是 Growth 首页的一次分析结果。



图 7-8 Growth Studio 首页 PageSpeed 分析示例

从图 7-8 可以看到，有红色的“高”字、黄色的“中”字和灰色的“低”字三种不同的级别，分别对应高、中、低三种不同的重要程度。单击对应的事项，如“启用压缩”，将会看到 PageSpeed 对相关建议的一些解释，以及存在问题的部分，结果如图 7-9 所示。



图 7-9 PageSpeed 建议和意见

从图 7-9 中可以看到，如果我们开启压缩，将会减少 233.5KiB 大小的文件传输，从而大大加快了页面的速度。我们的问题已经很明显了，在使用 Nginx 的时候“忘记”开启了文件压缩，但是我们需要怎么做呢？

好在这个工具提供了一站式的服务，我们只需要单击“了解详情”，就会跳转到相应

的网页做一个简单的介绍：

许多网络服务器可以通过调用第三方模块或使用内置程序将文件压缩为 `gzip` 格式，然后发送该压缩文件以供下载。这样在下载呈现网站所需的资源时，可以为您节省一些时间。

并且会针对性地给出建议：

您应在自己的网络服务器上启用压缩功能。以下这些参考展示了某些热门网络服务器上所启用的压缩功能。

- Apache：采用 `mod_deflate`。
- Nginx：采用 `HttpGzipModule`。
- IIS：配置 HTTP 压缩功能。

上面的建议里可以跳转到相应的配置页面。我们使用的是 Nginx，点击采用 `HttpGzipModule`，就可以跳转到 Nginx 关于启用 HTTP 压缩的页面：<http://wiki.nginx.org/HttpGzipModule>。在这个文档里给出了详细的说明和解释，以及一个 Nginx 的示例配置：

```
gzip                on;
gzip_min_length 1000;
gzip_proxied        expired no-cache no-store private auth;
gzip_types           text/plain application/xml;
```

我们只需要在 Nginx 中设置这些配置即可。默认的 Nginx 配置会开启压缩，只是其中的 `gzip_types` 并不全面如上述（代码所示）。因此，向 `gzip_types` 中添加压缩类型，如这里的 JavaScript、CSS、Jpeg 等文件。

```
gzip_types application/javascript text/plain text/css application/json
application/x-javascript text/xml application/xml text/javascript image/jpeg
```

在启用压缩之后，可以看到大小上的一些明显变化，即图 7-10 中的 **Size** 部分。



Name	Status	Type	Initiator	Size	Time	Timeline - Start Time
10.211.55.26	200 OK	document	Other	3.0 KB	435 ms	
bootstrap.min.css	200 OK	stylesheet	(index):15	24.9 KB	18 ms	
bootstrap.min.css	200 OK	stylesheet	Parser	118 KB	15 ms	
carousel.css	200 OK	stylesheet	(index):16	1.3 KB	18 ms	
carousel.css	200 OK	stylesheet	Parser	2.7 KB	16 ms	
style.css	200 OK	stylesheet	(index):17	571 B	18 ms	
style.css	200 OK	stylesheet	Parser	569 B	17 ms	
jquery-3.1.1.min.js	200 OK	script	(index):189	34.6 KB	20 ms	
jquery-3.1.1.min.js	200 OK	script	Parser	84.7 KB	19 ms	
bootstrap.min.js	200 OK	script	(index):190	11.9 KB	22 ms	
bootstrap.min.js	200 OK	script	Parser	36.2 KB	21 ms	

图 7-10 启用压缩后的文件传输大小

除了压缩文件，还有一些常见的网站性能优化事项。

2. 常见网站性能优化策略

针对网站的性能优化，我们可以从书上、博客里、网上轻松地找到各种相应的资料。因此，这里只列举出一些常见的性能优化策略及目的，并且这些优化措施主要都是针对开发人员罗列的。

(1) 减少 HTTP 请求

结合一些常见的减少 HTTP 请求事例，可以将其分为以下类。

- 合并 JavaScript 和 CSS。只是这种方式需要好好评估，因为合并过多的 JavaScript 可能会导致 JavaScript 文件过大。一个大的文件将增加加载时间，导致不好的用户体验。
- CSS Sprites。即将一个页面涉及的所有零星图片都包含到一张大图中。值得注意的是，像 Logo 这一类文件不要加到其中。
- 拆分初始化负载。将页面加载时需要的一堆 JavaScript 文件分成两部分：渲染页面所必需的和其他的。页面初始化时，只加载必需的，其余的一会加载。
- 划分主域。将资源划分的请求划分到几个不同的域上，以便加速资源请求。

当然，在浏览器端对应用进行缓存，也是减少 HTTP 请求的有效方法。

(2) 页面内部优化

HTML 页面内的优化的目的便是：尽快渲染出页面。常见的优化策略如下。

- 将 CSS 放在顶部，使浏览器尽早渲染出页面及其样式。
- 将 JavaScript 放在底部。如果有后台渲染机制，应该将 JS 放到页面底部来加速页面加载。如果是单页面应用，那么这个 JS 就应该在页面顶部。
- 压缩 HTML。在写模板的过程中，一些判断可能会导致页面有过多的空格。压缩这些 HTML，可以稍微提高页面加载速度。

这里的大部分内容都需要由修改代码来完成，或者也可以交由 PageSpeed 来完成。

(3) 启用缓存

我们将在本章后半部分详细说明：如何使用缓存来改善服务器性能。因此，这里简单罗列一下。

- 后台优化：如数据库端缓存。
- 页面缓存：如在应用层对页面进行缓存，或者使用缓存服务器来缓存页面。

(4) 减少下载量

简单地说，就是减少对服务器的请求：

- 使用 CDN。
- 使用外部 JavaScript 和 CSS。
- 使用 gzip 压缩。
- 缓存：添加 Expires 头、配置 ETag。

对单页面应用来说，我们还可以在浏览器缓存 Ajax 数据请求。

(5) 网络连接上的优化

主要就是对域名到服务器进行优化，因此，从方法上有：

- DNS 域名解析加速。
- 减少 DNS 查找。
- 使用 HTTP 2 来加速 HTTPS 请求。

上面有这么多的优化事项，真正要做起来就不是一件容易的事了。因此，为了减少这方面的时间花费，我们可以选择使用一些成熟的方案，如 PageSpeed 的 HTTP 服务器模块。

3. 使用自动优化工具

Google 也提供了 PageSpeed 相应的服务器模块可直接用于自动优化，它可以支持主流的 Nginx、Apache、IIS 服务器，我们只需要在编译的时候加入这个模板即可。PageSpeed 的服务器端模块可以省去优化 CSS、JS 和图片的过程，以及对 CSS 和 JavaScript 压缩、合并、级联、内联，生成一个新的 Script 和 CSS 文件。另外，还有图像优化：剥离元数据、动态调整、重新压缩，如针对 Chrome 浏览器生成 WebP 文件，还可以推迟图像和 JavaScript 加载等。

下面安装 PageSpeed 试试。需要注意的是，编译 Nginx 需要有一定的编译经验，在这个过程中会遇到一系列的问题。在安装之前需要先安装好依赖的软件包：

```
sudo apt-get install build-essential zlib1g-dev libpcre3 libpcre3-dev unzip  
libssl-dev
```

适用于 Nginx 服务器的 PageSpeed 模板叫 ngx_pagespeed，我们可以直接使用官方提供的安装脚本来实现自动安装：

```
bash <(curl -f -L -sS https://ngxpagespeed.com/install)  
--nginx-version latest
```

在这个安装过程里，它将会安装最新版本的 Nginx 和 ngx_pagespeed 模块，最后将

会生成 `nginx` 的命令文件，生成的文件路径位于 `/usr/local/nginx/sbin/nginx`。

由于不同的运维环境下对 `Nginx` 的配置是不一样的，因此，在安装过程中会提示是否添加自定义的配置语句，内容如下：

```
About to build nginx. Do you have any additional ./configure
arguments you would like to set? For example, if you would like
to build nginx with https support give --with-http_ssl_module
If you don't have any, just press enter.
>
```

这时可以添加我们的配置，如下是一个适合于之前代码的简单配置。

```
--sbin-path=/usr/sbin/nginx --conf-path=/etc/nginx/nginx.conf --with-http_
ssl_module
```

上面的配置中设计了生成的 `nginx` 二进制文件的路径，以及默认的配置文件路径，并添加了 `SSL` 模块。

随后，还需要对 `Nginx` 配置文件 `growth-studio.conf` 进行相应的设置，添加下面的代码：

```
pagespeed on;

# Needs to exist and be writable by nginx. Use tmpfs for best performance.
pagespeed FileCachePath /var/ngx_pagespeed_cache;

# Ensure requests for pagespeed optimized resources go to the pagespeed
handler
# and no extraneous headers get set.
location ~ "\.pagespeed\.([a-z]\.)?[a-z]{2}\.^[^.]^{10}\.^[^.]+" {
    add_header "" "";
}
location ~ "^/pagespeed_static/" { }
```

```
location ~ "^/ngx_pagespeed_beacon$" { }
```

上面的配置代码为 Nginx 开启了 pagespeed，并配置了相应的文件缓存路径等。我们还可以使用 EnableFilters 进行不同类型的配置定制，如转化 git 为 png:convert_gif_to_png、重新编译 PNG:recompress_png 等。更多详细的定制配置请参见官网的示例文档。

在重启服务之后，可以查看页面的结果，图如图 7-11 所示。

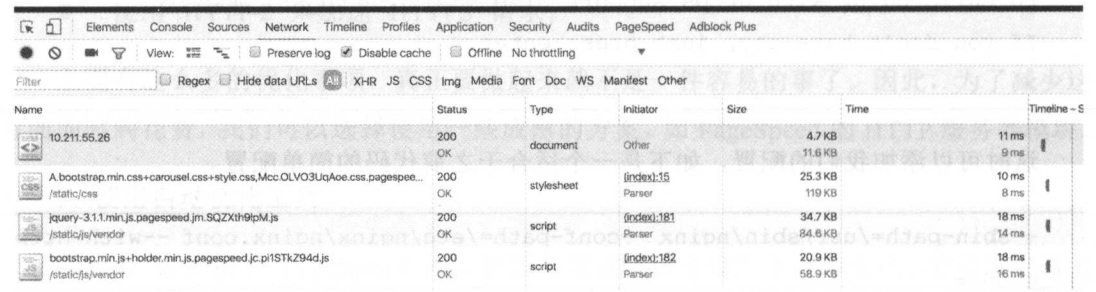


图 7-11 在页面中测试

图 7-11 中的第二、三、四个网络请求都是合并后的 JavaScript 和 CSS 文件。原先的三个 CSS 文件：bootstrap.min.css、carousel.css、style.css 就会合并为图 7-11 中的。

```
A.bootstrap.min.css+carousel.css+style.css,Mcc.OLV03UqAoe.css.pagespeed.cf.vKNcaXZtsI.css
```

同样，JavaScript 文件也会被合并为两个文件：

- jquery-3.1.1.min.js.pagespeed.jm.SQZXth9lpM.js
- bootstrap.min.js+holder.min.js.pagespeed.jc.pi1STkZ94d.js

这些文件会被 PageSpeed 模板压缩并存储到配置中写好的 FileCachePath 中的路径。除此之外，PageSpeed 模板还会帮助我们进行一系列的优化，现在可以重新用之前的 Chrome 插件进行测试，测试结果如图 7-12 所示。



图 7-12 重新测试结果

从图 7-12 可看出，我们还可以继续做优化，由于使用的图片字体文件没有设置缓存，打开相应的条目会看到：glyphicons-halflings-regular.woff2（失效时间未指定）。因此，我们可以通过向 /etc/nginx/mime.types 添加对应的类型来让 Nginx 设计缓存时间，如下：

```
font/woff2    woff2;
```

添加完后，再看看网页的打开速度等内容，就可以得到一个更高的分数。不过，这主要是因为这里的环境比较简单，要优化起来比较容易。当在一些更复杂的项目上使用时，就会遇到一些问题，如请求了第三方广告，或者有很多图片，这时就需要考虑使用延时加载这样的方案，这既可以让用户更快地打开页面，又可以完整地提高网络的功能。

当我们使用 PageSpeed 来分析、优化应用在网页中的性能时，主要集中于服务器返回网页，到网页加载完成的时间。这将向用户提供一个更好的用户体验，但是它能改善应用的性能是有限的，因此，需要寻求一些工具来帮助了解应用的性能。

7.2.2 后台优化：使用应用性能管理工具

在没有应用性能管理工具（APM，即 Application Performance Management 的缩写）时，如果需要对应用进行优化，就需要不断地调试、阅读源码才能找到问题。如果这是一个多人协作的项目，对项目进行优化的难度也会随着代码量的增加而不断加大。而了解应用性

能瓶颈的最好方法就是：查看程序中运行时间最长的部分。这时可以考虑使用性能管理工具来分析应用的性能。

这些性能管理工具运行在应用低一层的底层——语言层面，在应用运行的时候，它们的探针就会开始运行，并不断地收集应用的数据。这些数据将会被收集起来，发送给系统后台，这时可以在后台了解应用的运行状况。性能管理工具会分析应用的五个维度。

- **终端用户体验监控**，分析用户加载、渲染时间等有关于用户体验的事项。
- **应用运行时架构**，监控应用程序的所有节点和服务器等。
- **用户自定义的事务分析**，监控用户自定义的事务，或者一些与业务相关的 URL 页面定义等。
- **应用组件监控**，对应用程序的中间件进行监控。
- **报告及应用数据分析**，为应用程序提供度量和报告，并对其进行可视化。

同时，性能管理工具将使用应用性能指数（Apdex 的全称为：Application Performance Index）来衡量用户对应用性能的满意值。Apdex 定义了应用程序响应的最优时间为 T ，同时根据这个目标时间 T 定义了三种不同的性能表现。

- **满意**：应用响应的时间低于或等于目标时间（ T 秒），用户的工作不会因为加载时间过长而受阻。
- **可容忍**：用户感觉到响应滞后，响应时间大于目标时间（ T 秒），但是用户能忍受这个过程。
- **挫折**：响应时间大于四倍的目标时间（ T 秒），用户无法忍受这个过程，便会离开网页。

下面将 New Relic 作为应用性能管理工具来分析和展示应用程序的性能。

7.2.3 使用 New Relic 进行优化

New Relic 是国外知名的监控服务商，它可以实时地对应用进行监控和分析。我们选择使用 New Relic 的主要原因是，它提供了一个免费的、无期限基础版本。对一般的小型 Web 应用来说，这个免费版本就够用了，这个版本里包含了一般应用需要的：吞吐量、应用响应时间、错误报告、数据库度量等功能。专业版里提供了一些高级的功能，这些功能更适合于中、大型的 Web 应用，如部署追踪、服务地图、衡量负载报告等功能。

由于应用在线上的数据比较少，我将继续使用我的博客 (<https://www.phodal.com/>) 数据来展示：如何用 New Relic 进行分析。我的博客也是基于 Django 来开发的，因此，不需要担心与这里内容不符合的问题。博客每天的访问量在 500 人次左右，因此，数据本身也是足够用的。

1. New Relic 安装

在开始之前，希望读者已经注册好了一个 New Relic 账号，注册地址为：<https://newrelic.com/signup>。

登录 New Relic 之后，就可以集成 New Relic 到我们的应用里。New Relic 可以支持基于 Ruby、PHP、Java、.NET、Python、Node.js、Go 等语言或者技术的应用，我们使用的是 Python，选择完后就可以开始进行相关的设置，设置过程如下：

- 获取一个密钥。
- 再用这个密钥生成一个配置文件。
- 重新运行应用。

官网的设置步骤如图 7-13 所示。

1 Get your license key

Reveal license key

2 Install the New Relic Python agent

```
pip install newrelic
```

3 Generate the configuration using your license key

```
newrelic-admin generate-config <your-key-goes-here> newrelic.ini
```

4 Start the Python agent

See also: Python Agent Integration

Execute your WSGI server by adding some New Relic configurations like this:

图 7-13 New Relic 设置步骤

在网页端获取密钥，随后安装 newrelic 的库：

```
sudo pip install newrelic
```

再根据密钥生成相应的配置文件，命令如下：

```
newrelic-admin generate-config <your-key-goes-here> newrelic.ini
```

可以将配置文件放到项目的相应位置，然后就可设置这个环境变量，并运行程序：

```
export NEW_RELIC_CONFIG_FILE=newrelic.ini  
newrelic-admin run-program gunicorn -w 2 growth_studio.wsgi
```

这里的 `newrelic-admin run-program` 就会在应用与语言的底层之间创建一个钩子（Hook）来监听应用对函数的调用等内容。在完成设置的几分钟过后，就可以在后台看到

数据，并根据这些数据分析应用的情况。

2. 使用 New Relic 分析应用

现在我们可以打开 New Relic 的后台，在 APM 页面就可以看到应用信息的基本信息，如图 7-14 所示。

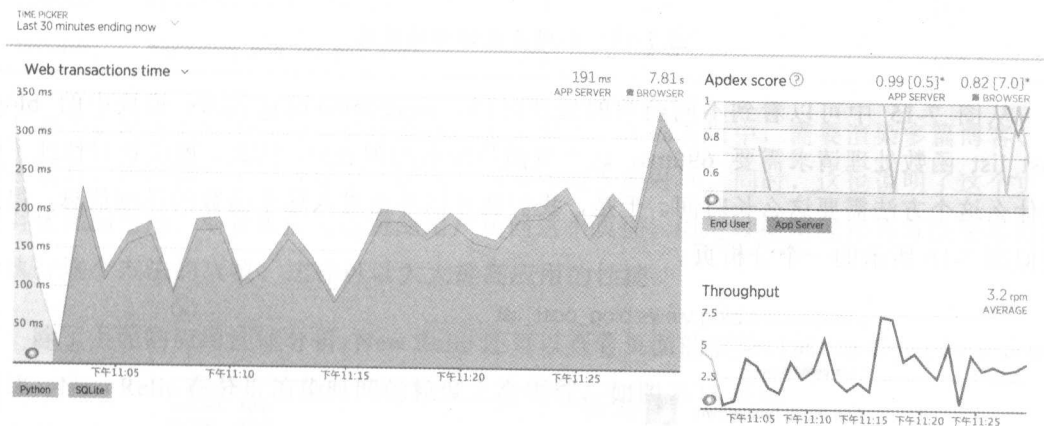


图 7-14 基本信息

图 7-14 中，左侧是服务器端处理的响应时间，即从应用接受请求，到返回 HTML 给 HTTP 服务器的时间。图 7-14 中分为两种颜色：较大的那部分是 Python 语言运行的时间，即运行应用逻辑所说的时间：平均每个请求需要 200 ms 左右。而数据库相关部分则需要花费大于 30ms。图 7-14 中右上部分显示的就是 Apdex 值，右下显示的则是网站的吞吐量。

对大型应用来说，其瓶颈应该是相反的，即处理数据库花费更长的时间，而应用花费的时间会更短。对我的博客来说，因为服务器性能的问题，所以运行逻辑代码的时间会比较长。

在概览的下方，我们就能看到页面的处理函数及服务器响应时间的关系，如图 7-15 所示。

Transactions	App server time	Error rate
/mezzanine.blog.views:blog_post_list	690 ms	0.00 %
/mezzanine.blog.views:blog_post_detail	268 ms	
/feed.view:blog_post_feed	251 ms	
/homepage.views:homepage	131 ms	
/amp.views:amp_blog_post_detail	98.8 ms	

图 7-15 不同函数的响应时间

从图 7-15 中可以看到不同函数的处理时间，mezzanine.blog.views 模块中的 blog_post_list 函数处理请求需要 690ms，这个页面是博客的列表页。因此，就能针对性地了解为什么这个方法需要这么长时间。于是，我们就可以单击进入这个函数的详细信息，得到类似图 7-16 所示的一个分析页。

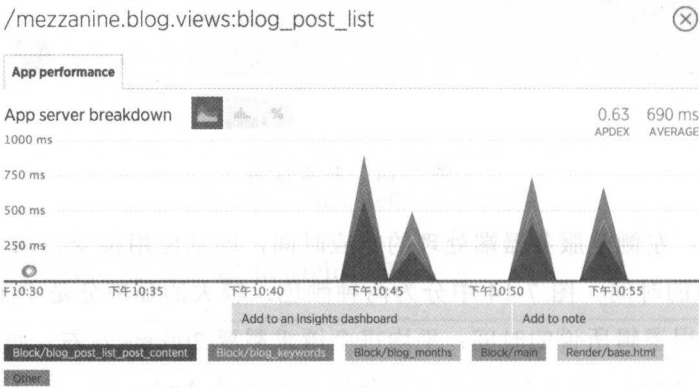


图 7-16 响应时间图表

从图 7-16 中可以看到，blog_post_list_post_content 模块花费了相当多的时间来执行。我们也可以看到一些更详细的信息，表 7-2 就是模板文件中不同函数的执行时间。

表 7-2

Category	Segment	% Time	Avg calls (per txn) Avg	time (ms)
Template	Block/blog_post_list_post_content	53.3	8.25	368
Template	Block/blog_keywords	13.7	1.0	94.6
Template	Block/blog_months	5.4	1.0	37.6

续表

Category	Segment	% Time	Avg calls (per txn) Avg	time (ms)
Template	Block/main	3.3	1.0	22.6
Template	Render/base.html	2.4	1.0	16.7
Database	SQLite blog_blogpost select	2.4	4.0	16.3
Template	Render/pages/menus/footer.html	2.3	4.0	15.7
Template	Block/blog_post_list_post_links	2.2	8.25	15.3.3

表 7-2 说明了应用时间主要花费在渲染页面上。在列表页中, 需要渲染多篇博客、关键字信息、按时间来分类等信息, 因此, 需要比较长的时间。同时, 这也说明了这个页面还有优化的空间。尽管我们无法避免用户访问这个页面, 但可以使用缓存等方法来尽量减少执行这个方法的次数。这样可以大大提高应用的性能。

除了上面的应用性能分析, New Relic 还可以查看页面的渲染时间。与 Google Analytics 相比, New Relic 在分析渲染时间的粒度上会更细, 如图 7-18 所示。

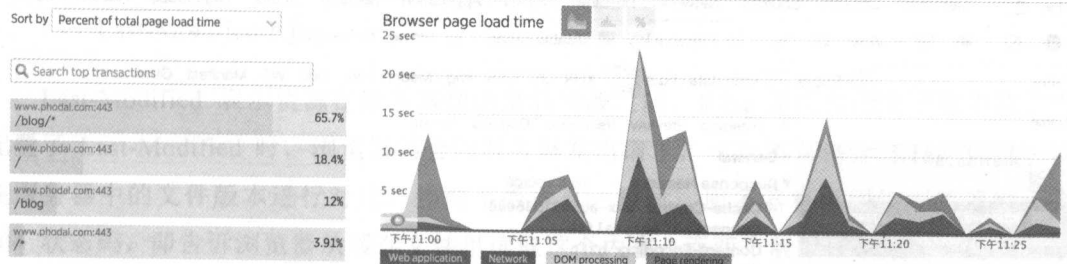


图 7-18 New Relic 页面渲染

图 7-18 中的加载时间分为以下四部分。

- 应用程序的处理时间。
- 网络传输时花费的时间。
- DOM 解析时间。
- 页面渲染的时间。

由于网络原因，这个渲染时间并不是那么准确。建议读者仅作为参考，或者关注类似 PageSpeed 这样的网页性能优化。

7.2.4 缓存初入

针对应用的性能问题，有一个很不错的措施就是采用缓存。

1. 浏览器缓存

在 Web 前端，我们可以采用一些缓存策略来加速用户打开网页的速度：静态资源缓存。即使用浏览器的缓存机制，将 JavaScript、CSS、图片等静态文件存储在本地。我们只需要使用 HTTP 服务器实施这种类似的缓存机制，如本章里使用的 ngx_pagespeed。我们可以通过设置 HTTP 头的 Cache-Control、Expires、Last-Modified 等字段来缓存，如图 7-19 所示。

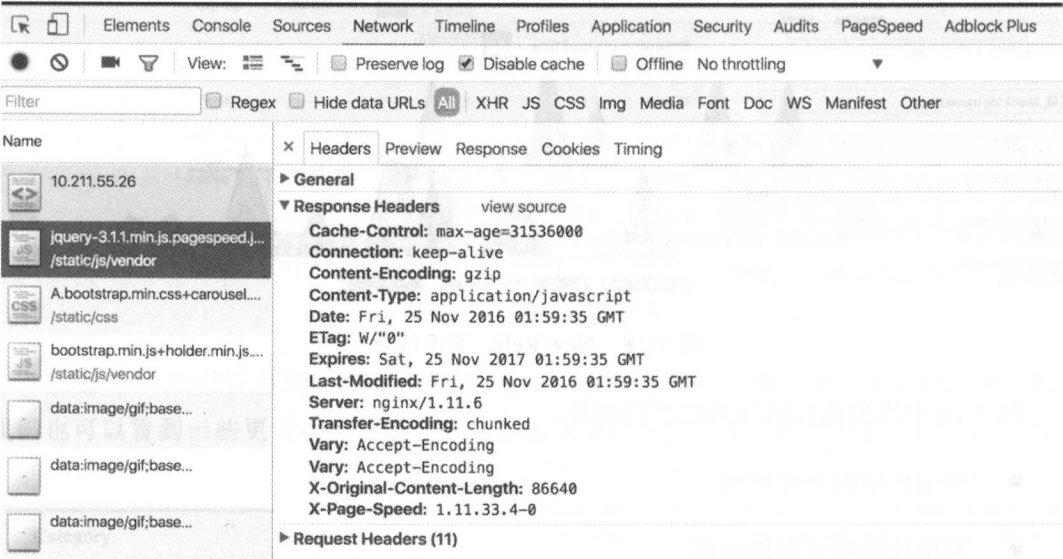


图 7-19 一个 CSS 文件的 HTTP 头信息

图 7-19 是使用 ngx_pagespeed 后自动设置的缓存，包含三种不同类型的资源缓存。

(1) Cache-Control

Cache-control 需要指定一个以秒为单位的 `max-age`，即缓存的最长时间。如 `Cache-Control: max-age=3600`，表示这个资源将在 3600s 后过期，即一小时内。我们再请求相应的资源时，将会直接使用本地的缓存，这样可以加快页面的渲染过程。这个时间是需要依据我们的需求来设定的，常见的缓存时间是一周或者一个月。同时，我们也可以设置 `public` 或者 `private` 来表明这个资源是为某个用户私有的。因此，通常会将这个参数设置为 `public`。

图 7-19 中的 `Cache-Control:max-age=31536000` 表明这个资源会缓存一年。

(2) Expires

Expires 会直接指明缓存的具体过期日期。在图 7-19 中，指定的过期时间是: `Expires: Sat, 25 Nov 2017 01:59:35 GMT`，即到 2017 年 11 月 25 日，这个资源才会过期。当 Expires 和 Cache-Control 同时出现时，以 Cache-Control 设置的时间为主。

(3) Last-Modified/ETag

Last-Modified 表示资源在服务器端的最后修改时间，ETag 是文件的唯一标识符。当配置了 Last-Modified 时，浏览器仍会向服务器发出请求，这个请求包含 ETag 信息，以便服务器中的文件版本进行对比。当两个版本号一致时，服务器将返回给浏览器端一个 304 状态码，即告诉浏览器从缓存文件里读取这个内容。而两个版本不一致时，则重新获取资源的新版本。

2. 应用缓存

在应用端，我们做缓存的目的主要是减少计算和查询，如：

- 缓存数据库的查询结果。
- 缓存磁盘文件的数据。
- 缓存某个耗时的计算操作。

这部分内容主要依赖于 Web 框架及其相关组件。下面以 Django 为例介绍不同的粒度应用层缓存。

(1) Memcached

Memcached 是 Django 中最快、最高效的缓存类型，它是一个完全基于内存的缓存服务。Memcached 缓存位于应用层与数据库之间，可以减少数据库读取负担，并将提供一个更快的读取速度。在上面使用 New Relic 对我的博客列表页的分析结果表明：我可以直接添加一个 Memcached。要添加 Memcached 很容易，只需要安装好 Memcached 和 python-memcached。然后在配置文件 settings.py 中添加相关的配置即可：

```
CACHES = {  
    'default': {  
        'BACKEND': 'django.core.cache.backends.memcached.MemcachedCache',  
        'LOCATION': '127.0.0.1:11211',  
    }  
}
```

(2) 数据库缓存

即将缓存保存到数据库中。我们只需要创建一个新的表，并在上面的 BACKEND 中配置即可。

(3) 文件缓存

即缓存保存到文件系统中。同样，只需要安装相应的依赖，以及修改 BACKEND 中的配置即可。

(4) 本地内存缓存

将缓存保存到内存中，适用于服务器运行不了 Memcached 时。同样，我们只需要安装相应的依赖，以及修改 BACKEND 中的配置即可。

(5) View 缓存

即针对不同的 View 函数设置缓存时间。由于要将缓存写在代码中，这种方式显得有

些不易控制。如下是一个简单的示例：

```
@cache_page(60 * 15)
def blog_list(request):
    return render_to_response('blog/list.html', {
        'blogs': Blog.objects.all()
    })
```

我们为博客列表页设置了 15s 的缓存时间。

(6) 模板系统缓存

我们只需要在模板系统中使用 `cache` 标签来注明哪些内容需要缓存，以及缓存时间，代码如下：

```
{% load cache %}

{% cache 600 blogcache %}
{% for blog in blogs %}
<div class="col-sm-4">
    <h2><a href="{{ blog.get_absolute_url }}">{{ blog.title }}</a></h2>
    {{blog.body | slice:"":80}}
    {{blog.posted}} - By {{blog.author}}
</div>
{% endfor %}
{% endcache %}
```

上面代码中的 600 是缓存时间：600s，`blogcache` 则是缓存的名称。

(7) 缓存 API

当我们只需要对特别的数据结果进行缓存时，就可以采用这种缓存方式。

3. 缓存服务器

缓存服务器运行在浏览器与原始服务器之间，如 Nginx + Gunicorn + Django 运行的

服务器，它可以看作是一个原生服务器。缓存服务器在服务器上缓存页面的内容，当我们第一个访问某个页面时，缓存服务器就可以保存这个页面的内容；在缓存期限内，当再次访问这个页面时，将由缓存服务器直接返回页面的内容，并且不需要经过原始服务器。因此，使用缓存服务器可以大大降低服务器的负载。

常见的缓存服务器有 Varnish、Squid，除此之外，在 HTTP 服务器中也可以使用第三方模块来扩展。由于 Django 本身提供了强大的缓存功能，并且使用缓存服务器可能会带来过多的问题。这里只做一个简单的介绍，有兴趣的读者可以详细了解。

7.3 小结

本章介绍了精益软件开发的另一部分：数据分析。我们需要先在应用上添加对应的监测服务，并添加对事件的追踪。随后，才能在后台了解到用户的信息、使用情况、转化率等情况。而这些数据可以帮助我们设计出更好的产品，我们所需要的就是从这些数据中去学习。除了介绍如何用流行的 Google Analytics 作为数据分析工具，我们还简单介绍了如何使用开源的监测工具 Piwik 来替代 Google Analytics。

另外，还介绍了如何使用类似的数据分析工具，用于分析应用程序及网页的性能：

- 借助网页性能测试工具 PageSpeed，我们可以得到一个相关的测试报告，并有针对性地网页进行优化。同时，也可以借助于 PageSpeed 的服务器端模块对网页进行自动优化。
- 借助应用性能管理工具 New Relic，可以轻松找到应用中的瓶颈部分，并能找到对应的瓶颈函数，同时对此进行特定的优化。

最后介绍了如何使用缓存来提供应用的性能。这些技巧在应用运行缓慢的时候尝试会带来意想不到的效果。

第 8 章将介绍如何进行持续集成。

优化书籍推荐

- 《网站性能监测与优化》。
- 《Web 性能实践日记》。
- 《Web 性能优化权威指南》。

第 8 章

持续集成与持续交付

本章将引入持续集成工具 **Jenkins**，并用该工具来完善项目的持续集成。同时，还将介绍与持续集成相关的工作流，并将开发更多的功能实践这些工作流来做好软件工程实践。最后，将使用这个工具来进行持续部署工作流的引入。

前面讲述的都是在本地上开发、测试、部署的环节，并且假定的开发者都只有一个人。当我们是一个人参与项目时，只需要在我们的机器上运行测试，然后在本地环境下完成打包、部署等工作即可。

但当我们是一个团队的时候，这些实践做起来就没那么容易了。我们每天都在向版本管理服务器上提交代码，这时就需要保证团队成员提交的代码都是正确的，即符合业务需求。而要保证代码是符合业务需求的，就需要编写测试。这样，当其他成员提交他们的代码修改时，我们就可以及时知道：他们是否破坏了我们的功能。如果没有测试作为保证，很难一下看到是哪个成员在什么时候破坏了哪些功能。

这时就涉及一个持续集成的概念，它是一种软件开发实践，即每天我们都把团队成员的代码集成在一起，并且要尽量保证每次集成的都可以正常工作。在这里使用的词是“尽量”，因为在真实的工作场景中，我们只能保证绝大部分的集成都是成功的。在实践较好的团队里，集成失败的原因大多都是由于一些环境因素引起的，如：操作系统故障、依赖出现问题、第三方服务失败等。

在了解、实践如何进行集成后，我们就可以考虑持续部署。虽然我们不一定有足够的权限去部署到生成环境，但我们可以做到的是自动部署到测试环境。对正常的开发流程来说，除了本地的开发环境和产品环境，还需要准备好测试环境(testing)、模拟环境(staging)。当向测试人员和业务人员展示我们的功能时，就需要在测试环境上展示。当我们想重现产品环境上的 Bug 时，就需要一个模拟环境。因此，如果可以保证从测试到部署的自动化，那么它将大大节省我们的时间。

在持续交付的过程中，我们的应用就像是流水线上的货品：不断地生产出来，经过工人测试，测试合格后打包、发布。

8.1 持续集成与 Jenkins

如果你对瀑布流有所了解，可能就知道：持续集成是为了解决瀑布流式开发带来的一

些问题的。即使是迭代式开发，即隔段时间发布一个新的版本，那么它也有可能采用的是类似瀑布流式的开发——在迭代的最后几天里集成代码、测试功能、修复 Bug，所有的问题都在这几天里爆发。这并不是因为某些人的开发问题，而是因为我们是在最后几天里集成代码的——我们将集成代码这个不可控制的因素都放到上线前的最后几天。

而持续集成就是为了解决关于代码集成的问题。当我们提交代码后，就自动按照设计的构建流程：执行 CheckStyle、执行单元测试、执行功能测试等，通过它帮找到代码中的问题。实现持续集成系统有以下关键的因素：

- 版本管理工具。
- 持续集成服务器。
- 测试环境。

在前面已经介绍了版本管理工具，也介绍了如何自动化部署应用到服务器上。加上缺失的持续服务器一环，我们就可以完成整个完整的流程了。而为了保证持续集成的功能，需要做到以下一系列的实践：

- 维护同一个代码源。
- 自动化构建。
- 支持自动化测试。
- 频繁提交代码，一天应该要提交一次，到多次。
- 每次提交都应该执行构建。
- 让构建尽可能快。
- 拥有接近产品环境的测试环境。
- 团队成员可以轻松访问。
- 团队成员都可以看到构建结果。

- 支持自动化部署。

这些都是我们在之前提到的一系列实践的集合。

8.1.1 工具选择与 Pipeline 设计

事实上，持续集成服务器就是一个运行着持续集成工具的服务器。这个服务器一般由两部分组成：主服务器（Master）和从服务器（即 Slave，或者代理服务器 Agent）。

- 主服务器：用于管理持续集成服务器、节点服务器、工具插件、设计流水线等。
- 从服务器：用于运行持续集成的流水线、返回运行结果等，接收主服务器的分配和管理。

当我们只有一个项目且团队成员不多的时候，可以直接使用一个机器既充当主服务器，又充当从服务器。而当我们有多个项目且团队成员较多、提交频繁时，就得考虑使用多个从服务器来运行构建。

1. 选择持续集成工具

当我们着手搭建持续集成环境时，首要问题就是选择一个持续集成工具——这与之前写应用是类似的，最开始的时候决定好框架，然后搭建构建系统等。一般来说，持续集成工具都会包含下列功能：

- 源代码控制系统。
- 依赖管理工具。
- 支持各种类型的测试。
- 可以使用插件来扩展系统。
- 支持流水线（Pipeline）。
- 可视化结果。

在现有的几个流行的持续集成工具里，它们在主要功能上都是相似的，只是不同的工具具有不同的侧重点。

Jenkins 是一个开源的、免费持续集成工具，其前身是 Hudson 项目，项目开始于 2004 年。它拥有丰富的插件、支持大多数的版本控制系、构建工具等，并且在网上很容易找到相关的资料。

Bamboo 是由 Atlassian 推出的一个收费的持续集成工具，它提供了更好的构建阶段、部署、构建流水线支持，提供了更好的商业支持。图 8-1 是 Bamboo 不同阶段的运行情况。

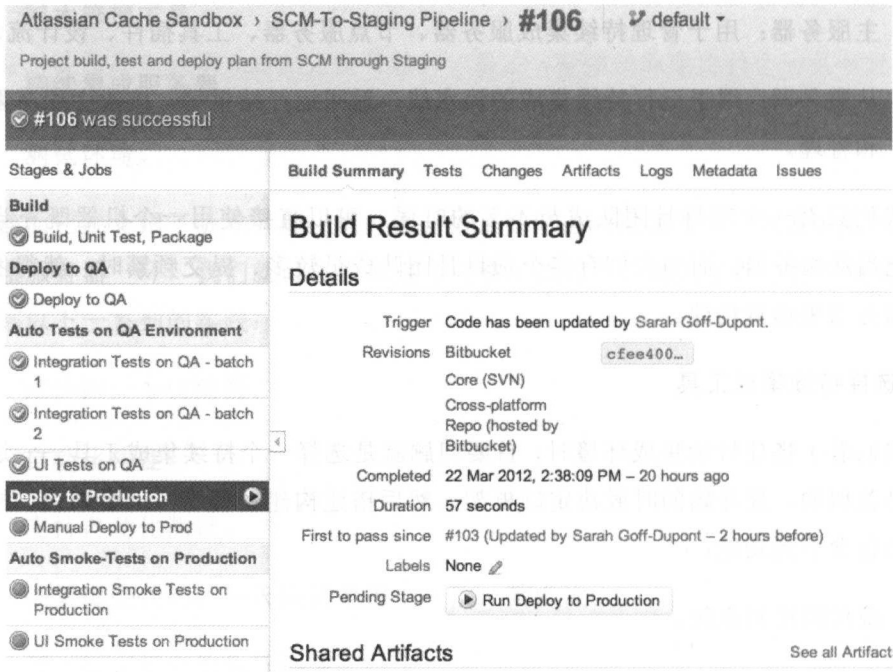


图 8-1 Bamboo Stage 示例

在图 8-1 的左侧部分，除了能看到持续集成工具正在运行的任务，还可以支持手动部署任务，即图 8-1 中的“Deploy to Production”阶段，我们只需要单击右侧的运行按钮即可。

GO-CD 是由 ThoughtWorks 推出的持续交付工具，它在持续集成的基础之上提供了更

好的持续集成、测试和部署，以及可视化构建过程。Go CD 更适合复杂一些的构建流程，图 8-2 是 Go CD 的流水线视图。

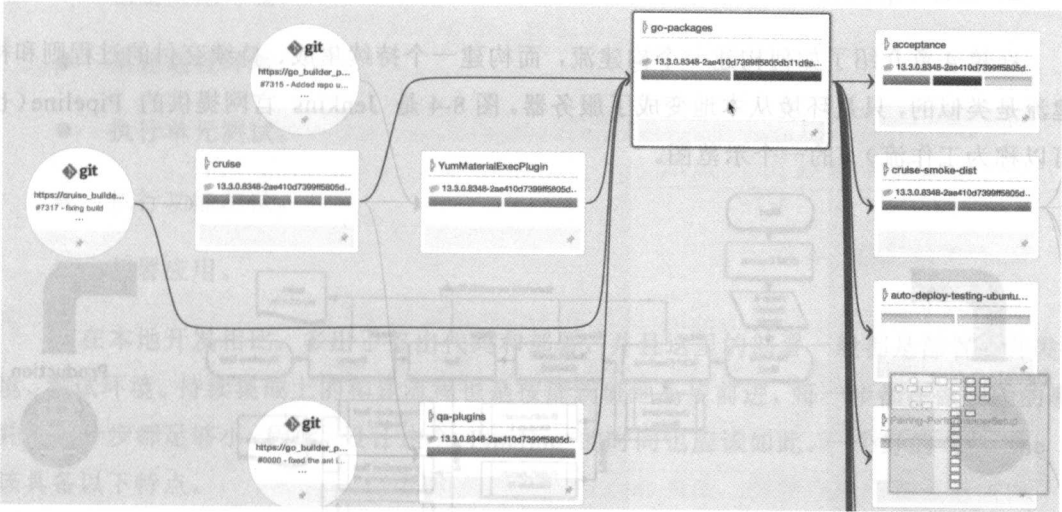


图 8-2 Go CD 流水线视图

由于 Jenkins 拥有更丰富的资料、免费使用，并且在新版本的 Jenkins 2.0 里已经逐渐添加了对流水线（Pipeline）的支持。图 8-3 是使用 Jenkins 设计的 Pipeline 的运行过程。

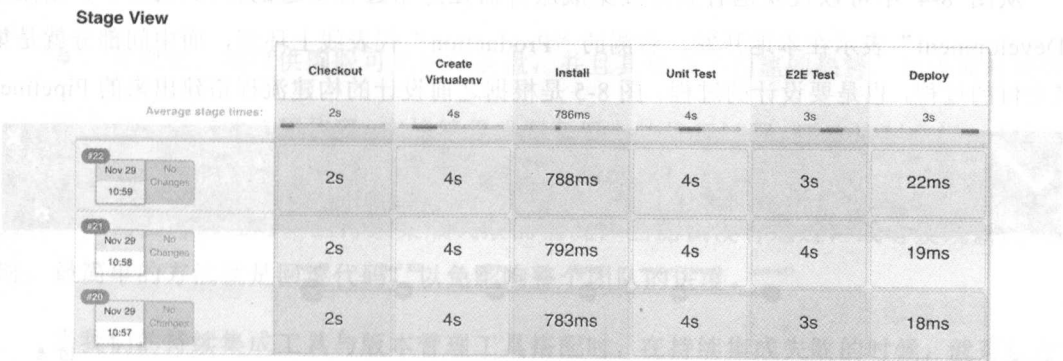


图 8-3 Jenkins Pipeline 时间视图

我们可以看到构建运行的不同阶段（Stage），如签出代码、创建虚拟环境、安装依赖、单元测试、功能测试等，并且也能看到不同阶段代码运行的时间。它也正在向更成熟的持续交付工具部署。

这里将以 Jenkins 为例来介绍使用持续集成工具。

2. 设计 Pipeline

在第 4 章介绍了如何构建一个构建流，而构建一个持续集成、持续交付的过程则和构建流是类似的，只是环境从本地变成了服务器。图 8-4 是 Jenkins 官网提供的 Pipeline（也可以称为工作流）的一个示范图。

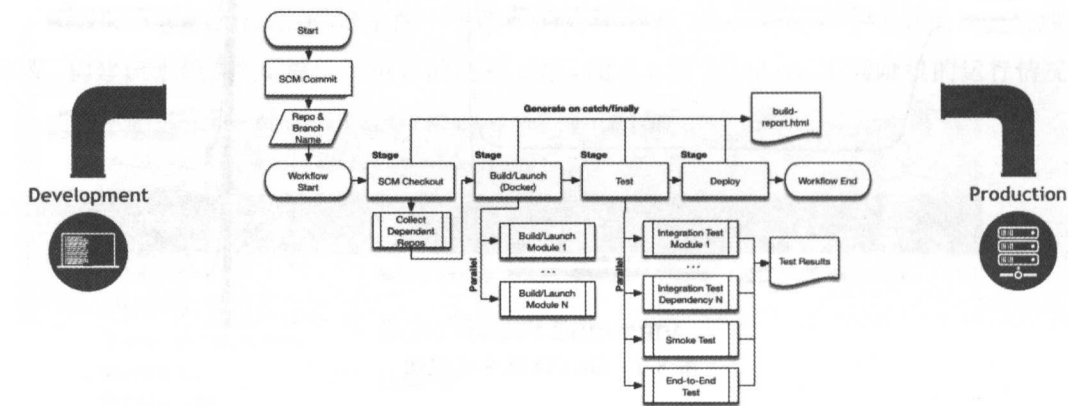


图 8-4 Jenkins Pipeline

从图 8-4 中可以直观地看到持续集成服务器在这个过程中起的作用。图 8-4 左侧的“Development”表示在本地开发，右侧的“Production”代表线上环境，而中间部分就是集成交付的过程，也是要设计的过程。图 8-5 是根据之前设计的构建流程搭建出来的 Pipeline。

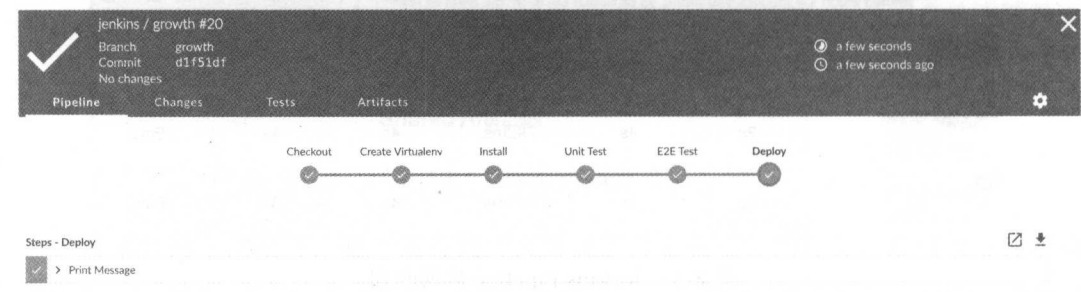


图 8-5 Growth Studio Pipeline

其步骤如下：

- 签出代码。
- 创建虚拟环境。
- 搭建运行环境。
- 执行单元测试。
- 执行功能测试。
- 部署应用。

与在本地开发相比，多出了签出代码和部署，并且这里的部署一般都是部署到开发环境、测试环境。持续集成上的构建流程也是按部就班地小步前进，每一步都在做明确的事，并且每一步都足够小。因此，设计这个 Pipeline 的时间也应该如此。一个好的 Pipeline 应该具备以下特点。

- 适应能力强：可以应对主服务器重启。
- 可暂停：在构建过程中可以等待开发人员输入或批准。
- 高效：可以从已保存的检查点重新运行。
- 可视化：能提供肉眼可见的仪表盘，并且其包含了构建的趋势，如当前时间等。

除了设计对应的流程代码、在持续集成服务器上的执行过程，还应该制定好一些规则，如：在提交代码前，应该先在本地运行测试。确保测试都是成功的，才向服务器提交代码。另外，还需要考虑应对一些持续集成失败的情况：当测试没有通过，或者发现新的 Bug 时，最简单的方法就是回滚代码，以免影响整个团队的进度。

当我们的持续集成工具与版本管理工具搭配时，在持续集成失败的时候，就可以在第一时间找到导致持续集成失败的人。而这时最好可以拥有一个额外的显示器，它用来实时显示持续集成的状态。这样我们就可以即时发现集成失败，并且找到相应的责任人来修复测试。如果修复这个测试需要相当长的时间，则可以考虑直接回滚代码。

如果不是人为原因导致的持续集成失败，而是诸如版本带来的环境问题，那么它应该

值得关注。有时，这些环境问题并不容易在短时间内修复，时间一长就会影响整个团队的持续集成进度。

下面让我们开始搭建这样的环境来了解持续集成。

8.1.2 Jenkins 搭建持续集成

1. 在本地机器上运行

要在自己的机器上使用 Jenkins 的简单方法就是到官网 (<https://jenkins.io/>) 下载最新的 jenkins.war 文件，然后运行这个文件即可。

不过，由于软件本身是使用 Java 编写的，因此需要安装一个 Java 的运行环境。同样，对不同操作系统的用户来说，安装 Java 的方式也是不同的。

- Windows 用户可以直接到官网下载 JDK，然后将 JDK 添加到环境变量中。
- Ubuntu 等 Linux 用户有一个比较简单的方式是直接安装 openjdk-8-jre（可以选择安装 Oracle Java，只是安装起来比较麻烦），同样，使用包管理工具来安装：
`sudo apt install openjdk-8-jre`。

接着，只需要运行下载的软件包，语句如下：

```
java -jar jenkins.war
```

便可以在本地的机器上启动 Jenkins。

2. 在服务器上安装

在 Jenkins 的官网上可以查看到不同的服务器的安装方式。这里仅以 Ubuntu / Debian 系统为例，只需要添加 Jenkins 的软件源，就可以安装 Jenkins：

```
wget -q -O - https://pkg.jenkins.io/debian/jenkins-ci.org.key | sudo apt-key  
add -sudo sh -c 'echo deb http://pkg.jenkins.io/debian-stable binary/ >  
/etc/apt/sources.list.d/jenkins.list'
```

```
sudo apt-get update
sudo apt-get install jenkins
```

启动 Jenkins 也需要一个简单的命令，就可以运行 Jenkins 的服务。

```
$ service jenkins start
==== AUTHENTICATING FOR org.freedesktop.systemd1.manage-units ====
启动“jenkins.service”需要认证。
Authenticating as: Phodal,,, (phodal)
Password:
==== AUTHENTICATION COMPLETE ===
```

在第一次启动 Jenkins 的时候，将会解压 Jenkins 到 home 目录的 .jenkins 文件下，并执行相应的安装过程，同时启动一个 Web 服务，默认的端口为 8080：

```
Running from: /Users/fdhuang/repractise/growth-ci/jenkins.war
webroot: $user.home/.jenkins
May 12, 2016 10:55:18 PM org.eclipse.jetty.util.log.JavaUtilLog info
INFO: Logging initialized @489ms
May 12, 2016 10:55:18 PM winstone.Logger logInternal
INFO: Beginning extraction from war file
May 12, 2016 10:55:20 PM org.eclipse.jetty.util.log.JavaUtilLog warn
WARNING: Empty contextPath
May 12, 2016 10:55:20 PM org.eclipse.jetty.util.log.JavaUtilLog info
INFO: jetty-9.2.z-SNAPSHOT
May 12, 2016 10:55:20 PM org.eclipse.jetty.util.log.JavaUtilLog info
INFO: NO JSP Support for /, did not find org.eclipse.jetty.jsp.JettyJspServlet
Jenkins home directory: /Users/fdhuang/.jenkins found at: $user.home
/.jenkins
May 12, 2016 10:55:21 PM org.eclipse.jetty.util.log.JavaUtilLog info
INFO: Started w.@68c34b0{/,file:/Users/fdhuang/.jenkins/war/,AVAILABLE}
{/Users/fdhuang/.jenkins/war}
May 12, 2016 10:55:21 PM org.eclipse.jetty.util.log.JavaUtilLog info
INFO: Started ServerConnector@733a9ac6{HTTP/1.1}{0.0.0.0:8080}
```

在这个过程中，会生成用于初始化安装的一个密码。这个过程会显示在启动应用时的控制台里：

```
*****
```

```
Jenkins initial setup is required. An admin user has been created and a
password generated.
```

```
Please use the following password to proceed to installation:
```

```
5fe9de863317439cb4b8cd6ff53f580d
```

```
This may also be found at: /Users/fdhuang/.jenkins/secrets/initialAdminPassword
```

然后，只需要访问 <http://192.168.4.12:8080/>，便可以进行初始化设置：

- 解锁 Jenkins，查看 Administrator 密码 `sudo cat /var/lib/jenkins/secrets/initialAdminPassword`。
- 进入“下一步”后，它将自动安装插件。
- 中途遇到 `This Jenkins instance appears to be offline.`，意味着需要代理才能安装插件，可以跳过这部分。

如果无法访问 Jenkins 的插件服务器，建议可以从本书的源码地址：<https://github.com/phodal/growth-code> 中找到这些插件的下载地址。下载插件完成后，需要做以下工作。

- 先停止 Jenkins 服务，并解压插件。
- 将插件复制到 `~/.jenkins/plugins` 目录下。
- 重新启动 Jenkins 的服务，就可以完成插件的安装。

Jenkins 的插件以 `.jpi` 文件结尾，安装插件的过程实际上就是在解压这些 `.jpi` 文件。

除 Jenkins 推荐的插件外，还需要安装下列插件。

- Pipeline Plugin：可以提供 Pipeline 支持。

- BlueOcean beta: 为 Jenkins 提供更好的用户体验。

安装完成后,会要求我们创建一个管理员用户。然后就可以开始使用 Jenkins 来创建我们的任务。

在登录首页后,就会看到“开始创建一个新任务”的提示,此时就可以创建一个新的任务,如图 8-6 所示。

图 8-6 Jenkins 创建项目

我们可以使用 Jenkins 创建不同类型的项目,自由风格、Maven 项目、Pipeline 等,也可以从一个现有的项目里复制。这里将先介绍通用的自由风格,再介绍使用 Pipeline 来创建项目。

在最上方中输入项目名,并选择“构建一个自由风格的软件项目”,就可以进入任务的配置页面,在这个页面里可以配置如下内容。

- 源码管理,配置源码的来源,如 git、svn 等。
- 构建触发器,配置触发任务及构建的条件,可以是远程触发、代码修性、定时构

建等。

- 构建环境，对构建环境的一些操作，如为构建日记添加时间戳、构建开发前删除工作区等。
- 构建，使用不同的构建脚本、工具等配置构建，可以使用 `shell`、`gradle` 脚本等。
- 构建后操作，在构建完成后执行的相关操作，能支持发送邮件、打包、触发其他构建等。

对我们来说，需要做以下步骤。

- 配置源码管理。
- 创建构建触发条件。
- 按步骤创建 `shell` 脚本。
- 添加构建完成的步骤。
- 测试任务。

首先，在源码管理中选择 `Git`，并填入代码的地址 `https://github.com/phodal/growth-studio`。然后构建触发器，一共有五种类型的触发器。

- 触发远程构建（例如，使用脚本）。
- `Build after other projects are built`——在其他项目构建完成后，执行这个构建。
- `Build periodically`——定期执行，如每日构建、每夜构建等。
- `Build when a change is pushed to GitHub`——当发生代码修改时。
- `Poll SCM`——定期从版本控制检测代码是否有修改，如果有则运行。

因条件限制，我们将在这里使用 `Poll SCM`，并且将这个值设置为 `H/5 * * * *`，即每 5 分钟运行一次。`Jenkins` 的调度任务的语法与 `crontab` 定时任务是相似的，使用五个由空格分隔的值来表示：分、小时、本月的天数、月份、星期，其对应的值如下：

①分钟 (0~59)。

②小时 (0~23)。

③日期 (1~31)。

④月份 (1~12)。

⑤星期 (0~6)。

同时,还会使用一些特殊的符号“*”、“/”和“-”、“,”,其含义如下。

- “*”代表所有的取值范围内的数字。
- “/”代表每的意思。
- “*/5”表示每5个单位。
- “-”代表从某个数字到某个数字,如在第五位的1~5,可以代表星期一到星期五。
- “,”分开几个离散的数字。

例如,我们可以创建一个从周一到周五、早上九点到下午六点、**每分钟的定时:

```
H/5 9-18 * * 1-5
```

不过,这里最好的选择是使用 Build when a change is pushed to GitHub, 它的原理是:
This job will be triggered if jenkins will receive PUSH GitHub hook from repo defined in scm section

即当 Jenkins 在监听 GitHub 上对应的 PUSH hook, 当发生代码提交时, 就会运行测试。因此, 我们需要将服务器部署到外网中, 同时要与 GitHub 集成好。

由于我们暂时不需要一些特殊的构建环境配置, 可以将这个放空。接着, 就可以真正配置构建。

3. 配置构建

这里需要添加的构建步骤是：`execute shell`，先让我们写一个简单的安装依赖的 `shell`：

```
PATH=$WORKSPACE/py35env/bin:/usr/local/bin:$PATH
if [ ! -d "py35env" ]; then
    virtualenv --distribute -p /usr/local/bin/python3.5 py35env
fi
. py35env/bin/activate
pip3 install fabric3
```

上面的脚本用于创建 Python 的虚拟环境：在不存在 `py35env` 目录时，便会调用 `virtualenv` 来创建一个相应的虚拟环境；随后将激活这个虚拟环境，并安装 `fabric3`，以便在后面安装依赖、运行测试等。

然后，我们可以先运行构建，确保这一步是可以通过的。图 8-7 是一次构建的结果。

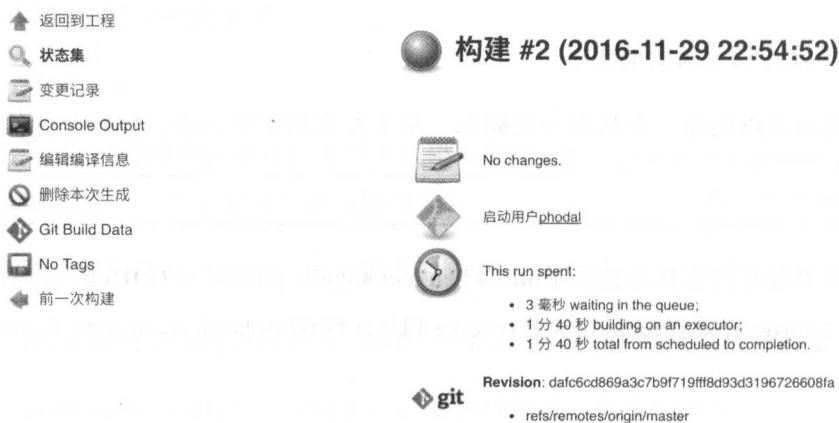


图 8-7 控制台输出

图 8-7 中指明了执行构建的用户、运行时间、相关的提交等信息，这些信息可以帮助我们在集成失败的时候，能更轻松地找到失败的原因。在图 8-7 左侧有一个“Console Output”，即看到终端的输出结果，在这里可看到详细的构建日记。

在编写 `shell` 的过程中，我们不断进行尝试，这其中会经历一些失败的情形——即使

有相关经验的程序员也会。图 8-8 就是一次编写构建脚本引起的构建失败的例子。

● 控制台输出

```
Started by user phodal
Building in workspace /Users/fdhuang/.jenkins/workspace/growth-test
Cloning the remote Git repository
Cloning repository https://github.com/phodal/growth-studio
> git init /Users/fdhuang/.jenkins/workspace/growth-test # timeout=10
Fetching upstream changes from https://github.com/phodal/growth-studio
> git --version # timeout=10
> git fetch --tags --progress https://github.com/phodal/growth-studio +refs/heads/*:refs/remotes/origin/*
ERROR: Error cloning remote repo 'origin'
hudson.plugins.git.GitException: Command "git fetch --tags --progress https://github.com/phodal/growth-studio +refs/heads/*:refs/remotes/origin/*"
stdout:
stderr: fatal: unable to access 'https://github.com/phodal/growth-studio/': Could not resolve host: github.com

at org.jenkinsci.plugins.gitclient.CliGitAPIImpl.launchCommandIn(CliGitAPIImpl.java:1745)
at org.jenkinsci.plugins.gitclient.CliGitAPIImpl.launchCommandWithCredentials(CliGitAPIImpl.java:1489)
at org.jenkinsci.plugins.gitclient.CliGitAPIImpl.access$300(CliGitAPIImpl.java:64)
at org.jenkinsci.plugins.gitclient.CliGitAPIImpl$1.execute(CliGitAPIImpl.java:315)
at org.jenkinsci.plugins.gitclient.CliGitAPIImpl$2.execute(CliGitAPIImpl.java:512)
at hudson.plugins.git.GitSCM.retrieveChanges(GitSCM.java:1054)
at hudson.plugins.git.GitSCM.checkout(GitSCM.java:1094)
at hudson.scm.SCM.checkout(SCM.java:495)
```

图 8-8 Jenkins 失败的构建

我们很容易遇到环境相关的问题、执行过程遇到的问题等，这时就更需要耐心来解决问题。

在上面运行的时候，我们是直接在 Jenkins 里配置运行脚本的。而这个运行脚本本身也属于基础设施的一部分，我们需要将它纳入版本管理器，因此，可以在项目里创建一个 ci 文件夹，在文件夹中放置不同阶段（stage）的构建脚本，如：

```
.
├─ deploy.sh
├─ e2e.sh
├─ install.sh
├─ setup.sh
└─ unit_test.sh
```

将上面的代码保存为 setup.sh，在提交代码后，就可以重新修改构建脚本为：

```
sh 'ci/setup.sh'
```

由于之前已经使用 Fabric 来完成大部分工作。因此，我们都可以直接使用 fab 命令来解决，在每一步里添加一个简单的脚本就可以，如下是 e2e.sh 的内容：

```
. py35env/bin/activate
fab e2e
```

同理，我们只需要将 fab 命令转化为不同的脚本即可，如 fab test 和 fab deploy。

在第一行里，我们激活了虚拟环境，然后执行运行功能测试的脚本。为了区分单元测试和功能测试，需要重新命名 test 文件夹，因此将其重新命名为 e2e。上面代码中的 e2e 是文件夹，可以直接修改脚本到 fabfile.py 文件中。

当我们使用独立的脚本时，就可以独立于持续集成工具，很容易切换到其他工具里。而当我们决定使用一个工具时，也可以选择一种更好的方式，如 Jenkinsfile。

8.1.3 使用 Jenkinsfile 简化流程

Pipeline as Code 是 Jenkins 2.0 推出的新功能，允许 Jenkins 用户使用代码定义流水线作业进程，从而实现对其的存储及版本化。它可以让 Jenkins 能够自动发现、管理和运行多个代码仓库和分支。而为了达到 Pipeline as Code 的目的，我们必须使用 Jenkinsfile。Jenkinsfile 是一个包含 Pipeline 脚本的容器，它详细说明了执行 Jenkins 作业所需要的特定步骤。

Pipeline 是基于 Groovy 的 DSL 来定义作业的，在 Pipeline 中包含了不同类型的关键词。

- 基本用法：诸如设置 stage、node、workspace 等。
- 文件系统：进行文件系统相关的操作，如修改目录、读写文件等。
- 流控制：进行构建流相关的操作，如输入、等待、重试等。
- Docker：进行 Docker 镜像相关的操作。
- 其他高级用法，如设置环境变量、加载其他脚本等。

这里由于篇幅所限，我们只介绍几个基本的用法，更详细的用法可以在使用的时候参考官方的文档。先来看一个简单的例子。

```
node {
    stage ('Checkout') {
        git 'https://github.com/phodal/growth-studio'
    }
}
```

在上面的代码里，一共有三个关键字 `node`、`stage`、`git`。其中的 `git` 就是用于克隆代码到本地，而 `stage` 和 `node` 则是用于定义构建的流程。

`node` 是所有的 Pipeline 必须要包含的内容，用于为 Jenkins 主服务器指明持续集成的节点。在 `node` 中的代码块就是这个节点所要执行的步骤，每个 Pipeline 都应该包含 `node` 相关的配置。如果 `node` 没有使用括号作为参数，输入所使用的节点机器，那么就会默认使用当前的机器。

`stage` 是任务在执行时在逻辑上不同的步骤。Pipeline 的语法主要由多个 `stage` 来组成，每个 `stage` 中包含了多个执行任务的步骤。如上面看到的签出代码、创建虚拟环境、单元测试等都应该在不同的 `stage`，它们之间是按顺序来执行的，并且是按定好的过程来执行，同时也用于标记不同的步骤。

在上面的代码里，在 `stage` 后的括号里使用 `Checkout` 来指明这个步骤的目的，即用于签出代码。然后在这个 `stage` 的代码块里使用 `git` 和源码来签出代码。

由于已经使用脚本来简化流程，因此可以很容易地按我们的步骤写出下面的代码。

```
node {

    stage ('Checkout') {
        git 'https://github.com/phodal/growth-studio'
    }

    stage ('Create Virtualenv') {
```

```
sh './ci/setup.sh'
}

stage ('Install') {
  sh './ci/install.sh'
}

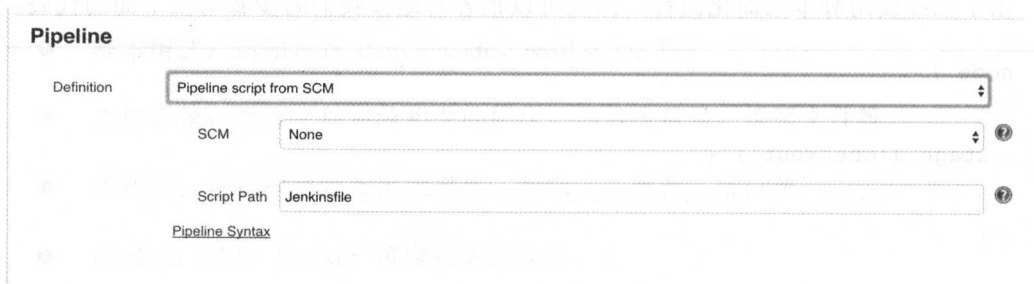
stage ('Unit Test') {
  sh './ci/unit_test.sh'
}

stage ('E2E Test') {
  sh './ci/e2e.sh'
}

stage ('Deploy') {
  sh './ci/deploy.sh'
}
}
```

在这个步骤里一共分为六步。除了签出代码，每一步都由一个简单的脚本构成。这样做的好处在于，即使我们切换了不同的持续集成工具，也可以顺利地执行代码。

剩下的只需要新建一个任务，并选择 Pipeline 作为我们的风格。然后，就可以使用 Jenkinsfile 作为构建流程，并且不需要其他配置，如图 8-9 所示。



Pipeline

Definition: Pipeline script from SCM

SCM: None

Script Path: Jenkinsfile

[Pipeline Syntax](#)

图 8-9 新建 Pipeline

同时借助于一些插件，如 Pipeline Stage View，可以在构建的时候看到更直观的构建过程，图 8-10 是 Stage View 插件的截图。

Stage View

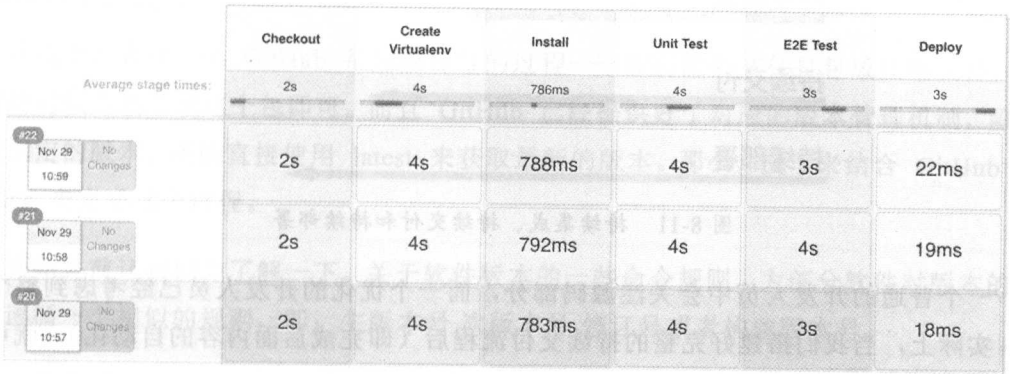


图 8-10 Pipeline 不同阶段视图

从图 8-10 中可以看到不同 Stage 的运行时间、每一步的运行情况，以及在某一个阶段失败的情况等。

当我们实现了对代码的持续集成之后，就可以向下一个步骤迈进：持续交付。

8.2 持续交付与持续部署初探

持续交付是持续集成的更高阶段，我们在持续集成的时候，考虑的是如何提高代码的质量，使用快速集成来改善软件集成流程。随后，应该关注于从集成完成到自动发布的过程，从提交代码到自动发布软件过程，就是持续交付。完成持续交付之后，就可以自动化部署软件到测试环境，但是这并不是持续部署。持续部署最大的挑战在于：部署是直接部署到产品环境。图 8-11 是三个持续之间的区别。

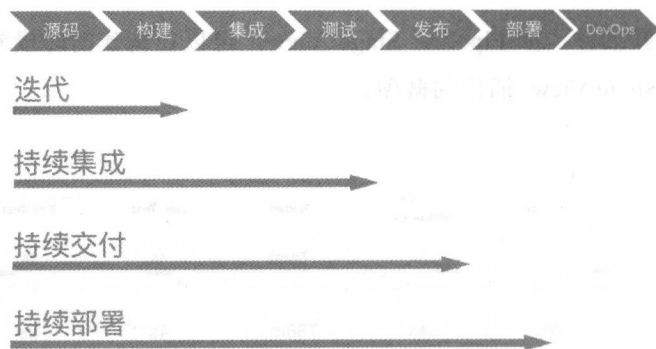


图 8-11 持续集成、持续交付和持续部署

一个普通的开发人员中会关注源码部分，而一个优化的开发人员已经考虑到整个部署。实际上，当我们搭建好完整的持续交付流程后（即完成后面内容的自动化），就可以关注迭代部分的内容。

持续交付与持续部署有一个很大的区别是：持续交付中的部署是手动控制的、可选定版本的自动部署到产品环境，而持续部署则是完全自动部署到产品环境。

8.2.1 持续交付

持续交付要做的就是：在任何时候的修改都是可以在任何时候部署的。而要做到持续交付并不是一件容易的事，它意味着一旦我们提交了代码，在持续集成服务器运行完后，就相当于发布了一个新的版本。这听上去有点骇人听闻，如果你不小心提交了一个 Bug，那么新的版本里就会带有这个 Bug，并且这个带有 Bug 的版本就会运行在测试环境中。在这个过程里，要避免 Bug 产生的最好方式就是在编写代码时编写测试。

如果我们对软件工程实践得好，那么在新版本与上线之间，应该只需要一个按钮就够了。而我们已经有了自动部署的能力，只需要补上自动发布版本和对测试环境进行自动化测试即可。

1. 基于 Jenkins 与 GitHub 的自动发布

在第 4 章已经提到了相当多的关于自动化构建的内容，甚至还提及如何基于 GitHub

来做发布管理。即，我们只需要执行下面的命令，就可以发布一个新的版本到 GitHub 上：

```
$ git tag v0.0.1
$ git push origin v0.0.1
```

在这个过程中，由 GitHub 帮完成打包的过程——即将源码文件打包成压缩文件，在部署的时候直接下载这个包即可。而且 GitHub 已经提供好了对应的版本管理机制，除了下载制定的版本，还能直接使用 `latest` 来获取最新的版本。那么，接下来结合 GitHub 和 Jenkins 来完成这个过程。

开始之前让我们先了解一下，关于软件版本的一些命名规则。大部分软件对版本的命名都遵循一个相似的规则，即：主版本号.次版本号.修订号或者构建版本号。

一般来说，这三个版本号都对应不同程度的修改。主版本号的修改意味着重大修改，新的版本可能已经不兼容旧的 API。而次版本号则是添加了新的功能。构建版本号或者修订号则表示一些小的日常修改。如果开发的系统并不对外开放，那么这种版本号的意義不大，甚至在很多时候会造成额外的困惑，如这里为什么会直接升级（一个大版本等）。

当我们在团队里使用敏捷及迭代号进行事务管理时，也会将迭代号作为软件的主版本号。这里推荐的版本做法是使用**迭代号.构建号**的命名方法。在每个迭代结束的时候，我们都会发布一次，如现在是迭代 20，而持续集成上的软件版本号是 200，那么发布的版本就变成 20.200。在下一个迭代发布的版本可能会变成 21.215。我们就很清楚线上运行的是哪个版本，同时能很好地帮助我们找到某个迭代对应的 Bug。

持续集成工具在执行构建的时候，都会有对应的版本号等参数。我们只需要取出相关的参数，同时在每次迭代开发的时候修改迭代号，就可以实现对软件版本的管理。如在 Jenkins 里，可以获取到一个名为 `BUILD_NUMBER` 的环境变量，这个环境变量是当前构建的号，我们只需要在打标签的时候加上版本号即可，如：

```
git tag 'v1.${env.BUILD_NUMBER}'
git push origin 'v1.${env.BUILD_NUMBER}'
```

由于我们无法直接在 shell 脚本中获取相关的环境变量，因此将其添加到的 Jenkinsfile

中，如下：

```
``bash stage ('Release') { sh "git tag -a 'v1.env.BUILD_NUMBER'-m'AutoTag:1.
{env.BUILD_NUMBER}'" sh "git push origin 'v1.${env.BUILD_NUMBER}'" }
```

每次当我们构建一个新的版本时，都会在 GitHub 服务器上创建一个新的 Release，它就可以直接用来安装。即，使用之前的脚本：

```
stage ('Deploy') { sh '. py35env/bin/activate' sh "fab deploy:
'1.${env.BUILD_NUMBER}'" } ``
```

如果要在持续集成服务器上完成同样的事，我们在持续集成服务器配置一个 GitHub 账号或者 Key，让这个 Key 可以访问我们的项目。需要注意的是，配置的时候要注意相关的安全设置——这个持续集成服务器拥有访问代码的权限。

2. 验收测试

当我们在非生产环境部署代码时，需要验收测试来帮我们测试与业务相关的功能。验收测试与集成测试的不同之处在于，验收测试是以功能的角度来对代码进行测试的，而集成测试则专注于测试系统的集成。事实上，这种测试和本地运行的集成测试是相似的，我们甚至应该考虑直接使用本地的测试作为验收测试——除了使用不同的参数和不同的测试链接。我们需要在测试里修改对应的链接，在之前的测试里，测试用的 URL 是本地的地址：

```
def test_can_visit_homepage(self):
    self.selenium.get(
        '%s%s' % (self.live_server_url, "/")
    )
    self.assertIn("Growth Studio - Enjoy Create & Share", self.selenium.
title)
```

在对服务器进行测试的时候，将其修改为服务器的地址：

```
def test_can_visit_homepage(self):
    self.selenium.get('http://10.211.55.26/')
```

```
self.assertIn("Growth Studio - Enjoy Create & Share", self.selenium.  
title)
```

同时，还会有其他一些变化，如：在本地测试时，为了方便直接调用 `model` 来创建博客和用户。而在服务器上就需要一些额外的准备工具，比如创建用户。这时需要编写相应的生成脚本，或者使用 `SQL` 来创建用户。如下是使用 `Python` 来创建一个超级用户的例子。

```
@task  
def prepare_ac():  
    with cd('growth-studio'):  
        with prefix('source ' + virtual_env_path):  
            run('echo "from django.contrib.auth.models import User; User.  
objects.create_superuser(%s, %s, %s)"'  
                ' | python manage.py shell' % ('test', "test@phodal.com",  
                "test"))
```

当我们需要创建博客的时候，就需要登录、添加博客，随后才能进行一堆的测试。同时，我们还需要准备好一份额外的数据库环境。准备好后，最后交付的流水线如图 8-12 所示。

Stage View

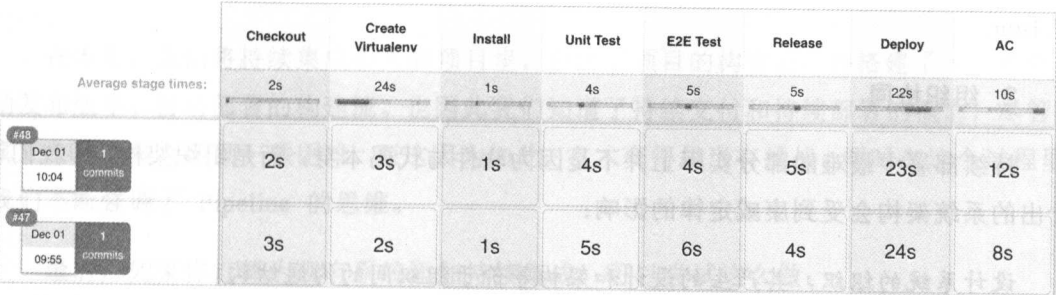


图 8-12 最终的 Stage View

单元测试完成后，运行集成测试，接着发布应用，然后部署，再运行验收测试。由于验收测试与集成测试做了同样的事，因此，一般会考虑在本地的集成测试中包含验收测试的所有功能。如果编写集成测试和验收测试是相同的开发人员，就应该考虑使用同

一份代码。两份代码会带来更多的问题，特别是业务发生变化的时候，要应对变化就变得更加困难。

8.2.2 持续部署初探

持续部署相当考验团队的协作能力——当我们提交完代码十几分钟后，应用就上线了。在这个过程中，一切都是完全自动化的，自动化测试、自动化打包、自动化发布、自动化部署等。自动化发布、部署、打包实际上都是大部分团队所能做到的，也许你也会说自动化测试也很容易。

1. 代码质量

要做好持续部署中的自动化测试并不是一件容易的事。这就意味着，开发人员一开始就要编写测试——而这在国内的互联网环境下是一件很难的事。如果你向一个在国内知名互联网企业中的开发人员咨询，他们是否会在编写业务代码的同时也编写测试？答案基本上是否定的。而持续部署除编写测试之外，还需要达到相当高的单元测试覆盖率，至少也要达到 90% 的要求。除此之外，还需要编写集成测试等。

代码有了充分的测试之后，我们才会在每次上线的时候有把握，才不会担心线上会出现 Bug。

2. 组织协同

持续部署中最难的部分实际上并不是因为软件与代码本身，而是组织架构。即我们设计出的系统架构会受到康威定律的影响：

设计系统的组织，其产生的设计和架构等价于组织间的沟通结构。

传统上，开发人员、测试人员、运维人员都属于不同的团队，每次发布的时候都需要根据不同的部门、组织来进行调整。而持续部署则意味着这几个部门的人是一起工作的，当开发人员完成一个功能时，测试人员就进行测试，并且自动上线。运维人员会在开发的过程中帮助开发团队来解决自动部署的问题等。而我们知道跨部门协作并不是一件容易的

事，要合并不同部门的人在一起更不容易。

3. 一些不足

虽然在上面的内容里已经讨论了相当多与环境相关的配置问题，然而仅仅只有这些内容是远远不够的。我们所涉及的只是环境相关的内容，缺少一些操作系统的基础设施的配置，也没有考虑使用云服务，我们的项目也相当简单——遇到复杂的项目时，就会遇到更多的问题。

我们使用的是 Django 框架，自带数据库迁移的功能，使得我们很容易就可以进行数据迁移。而当我们使用其他框架时，就需要考虑自己编写相应的数据库脚本，这时持续部署可能就更容易出错，并且在每次部署的时候，我们还应该对数据库进行备份。

我们还需要在持续部署的过程中考虑使用蓝绿部署来的机制，来保证部署的过程中不会影响用户的使用。

关于这些内容，建议读者在实际生产过程中遇到具体的问题再具体分析。

8.3 小结

在本章，我们将持续集成引入到项目里，设计了项目的构建流，并搭建了一个可以自动发布版本、自动部署的构建流。也因此我们知道了持续交付和持续部署的概念，尽管要真正实践这些会遇到一些困难，而这因为如此，它才变得相当有挑战。同时在这个过程里，我们不断强调了 Pipeline 的思维。

最后，如果你打算为现有系统添加持续集成，可以尝试这么做：

①创建时间间隔的自动构建。先采用每周或者更短时间的构建来引起学习，再不断地降低时间间隔（如每日），最后才是每次提交。

②添加自动化测试。在这个过程中，我们会遇到一个很大的挑战：说服开发人员写测试。

③提高测试覆盖率。当我们开始编写测试时，就已经有一个很好的开始，随后就是不断提交测试覆盖率，才能保证持续集成的效果。

④添加自动化验收测试。

⑤自动发布。

⑥测试环境自动部署。

建议读者尝试为过去编写的项目实践这些过程。

参考书籍及推荐阅读

《持续交付》《Jenkins 权威指南》

第 9 章

移动 Web 与混合应用

本章将介绍前端开发的一些基本要素，以及如何选择一个前端框架。还将引入混合应用框架 Ionic 2，使用它来开发混合应用，同时也作为我们的移动版本。在这个过程中，我们还将学习如何为现有应用创建 API。

当我们的应用上线一段时间后，我们在缝缝补补中度过——修复了一个大的 Bug，又引入了一个小的 Bug。而随着我们对统计数据的分析发现越来越多的用户使用移动设备来访问网站。这时就会有一个问题，我们是否创建移动版本？是否创建移动应用？

因此，在本章将介绍下列内容：

- 了解什么是单页面应用。
- 介绍单页面应用所需要的知识。
- 了解 Ionic 2 混合应用框架。
- 了解 RESTful API，并创建博客应用所需要的 API。
- 使用 Ionic 2 和 Angular 2 来创建博客应用。
- 了解 JSON Web Token，并搭建一个相应的 API。
- 编写应用的登录和注销功能。
- 了解如何使用 Ionic 2 来构建、发布应用。

因此，本章要涉及的知识和内容会比较多，建议读者可以阅读一些相关的资料。现在先让我们来了解一个单页面应用。

9.1 移动 Web 与单页面应用

早期的移动 Web 只是面向移动设备创建了一套新的模板，并且只在上面显示一些简单的内容。其采用的仍然是传统 Web 应用的模式，当用户点击链接时，仍然需要从服务器获取模板，再显示到本地的网页上。而随着移动设备性能的提高，以及移动浏览器性能的改善，人们便开始探索新的移动 Web 的模式，这就是单页面应用。

单页面应用，即 Web 应用运行一个页面里集成了系统的大部分功能，并且提高了类

似于桌面应用的用户体验。通常，当应用第一次运行的时候，就会加载页面所需要的 JavaScript、CSS、HTML，往后只需要加载所需要的资源，如数据、图片等。当用户进行操作时，页面会被重新沉浸，但是不会重新加载；当用户执行与和网络相关的操作时，将会由 JavaScript 动态地向服务器获取、上传数据。

这意味着，我们编写应用的重心将由后台转换到前台。我们需要编写大量的 JavaScript 代码，好让 Web 应用在整个过程中一直在浏览器上运行；同时，还需要对后台进行扩展，以便能拥有适合的 API 供前端调用。因此，当我们试图去编写一个单页面应用时，不仅要掌握好前端相关的技巧，还要懂得如何设计后台 API。

今天我们会发现大部分产品既有移动 Web 版本，又有对应的移动应用版本——与移动 Web 版本类似，只是一个手机应用，可以脱离浏览器独立运行，同时提供一些额外的功能。对采用移动 Web 与混合应用来构建产品的团队来说，他们会使用同样的技术栈来复用代码，如 React 和 React Native，又或者是 Angular 及基于 Angular 的 Ionic。他们分别针对不同情况下的用户使用，如：

- 移动 Web 版本，针对不想下载 App 的移动用户，或者是想了解网站应用的用户。
- 混合移动应用，针对将应用作为日常使用的用户。

混合应用与原生的移动应用相比，混合应用在性能上有一定的劣势，但是其跨平台能力给他带来了巨大的优势——只需要编写一份代码，就可以在不同操作系统的手机上运行，同时减少开发成本，并且由于开发过程中可以直接在桌面浏览器上调试。因此，在开发速度上更加高效。然而，如果你开发的应用对性能有一定的要求，就需要好好考虑。

由于移动 Web 和混合应用的技术栈上是类似的，并且代码可以通用。因此，我们将在本章使用混合应用作为示例，这些代码可以很容易地运行在移动浏览器上，并作为移动 Web 版本来运行。

在开始深入了解混合应用与移动 Web 之前，先对单页面应用有一个大致的了解。

9.1.1 单页面应用入门

在创建的博客应用里，为了显示一个博客，我们需要定义路由，才能跳转到相应的博客页面；需要编写逻辑代码，才能将它从数据库中取出来；需要编写显示模板，才能将它展示到页面上。在这个过程中，我们定义了路由，编写了控制器逻辑，美化了模板。因为 Django 是 MVC 框架，因此在实现上与一般的 Web 后台框架并没有太大的区别。

对编写前端的单页面应用来说，我们也需要类似的前端框架，它可以大大改善开发效率。这些单页面应用做了很多与后台 MVC 框架相似的工作。除此之外，它还需要进行复杂的诸如事件处理、DOM 修改等工作。

1. DOM 与事件

在进行前端设计的时候，我们需要了解 DOM 与事件。DOM (Document Object Model) 即文档对象模型，作为 HTML 和 XML 文档的编程接口而存在，并且独立于平台、操作系统与编程语言。HTML 文档中的所有内容都是节点，一个 DOM 模型则作为一个结构化的节点组而存在，并且在这个节点组中包含了属性和方法的对象。我们可以通过 JavaScript 来修改 DOM 中的内容，而 DOM 便是连接 JavaScript 和 HTML 的桥。

我们来看一个简单的例子。下面是首页中的 HTML 代码：

```
<div class="container">
  <div class="carousel-caption">
    <h1>《Growth: 全栈增长工程师指南》</h1>
    <p>帮助你构建知识体系，这也是其他技术书籍所欠缺的。它可以告诉你，可以学习什么，
    然后看什么书。</p>
    <p><a class="btn btn-lg btn-primary" href="https://github.com/
    phodal/growth-ebook" role="button">立即阅读</a>
    </p>
  </div>
</div>
```

我们可以用浏览器访问首页，并通过下面的代码访问这个节点：

```
var container = document.getElementsByClassName('container')[2]
```

接着，可以很容易通过这个节点来访问页面的 DOM，如下面的代码可以获取这个节点下的文本内容，我们就会得到上面 HTML 中的所有文本内容：`container.textContent`；也可以通过 `container.parentNode.parentNode.parentNode.parentNode` 一直向上访问节点，直至根节点。这是因为 DOM 是以节点树的形式而存在的，如图 9-1 所示。

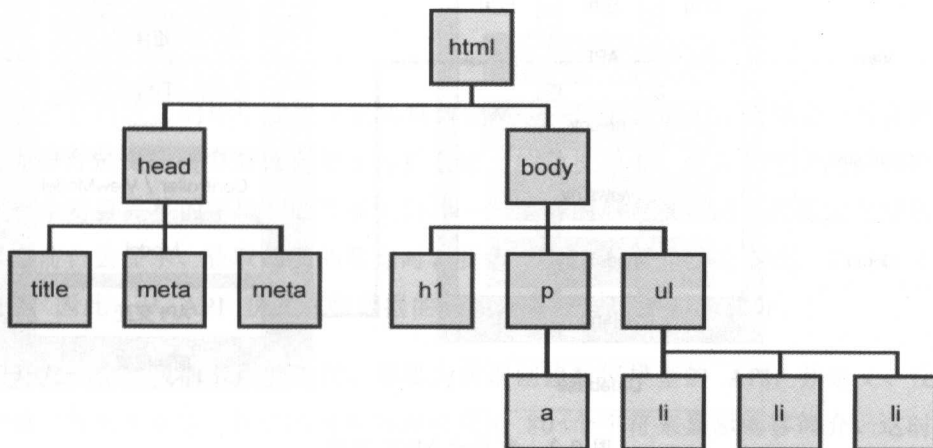


图 9-1 一个典型的 DOM 树

我们只需要知道这棵树一个叶子的存在，就能推导出整棵树。而 DOM 除了能让我们访问、修改网页中的内容，还可以创建相应的事件来响应用户的操作。当用户在页面上进行一些操作时，需要获取监听相应的 DOM，并创建相应的响应事件。如下是一个简单的登录按钮的监听事件：

```
var button = document.getElementById('login');  
button.addEventListener('click', function(){alert('Hello world');}, false);
```

我们先获取页面中 ID 为 login 的按钮，并为它创建一个点击事件的监听。当用户单击这个按钮时，我们的应用就可以对此做出响应。除此之外，还有一些诸如 `onkeypress`、`onkeydown`、`onchange`、`onselect` 等常用的事件，可以帮助我们监听用户的行为，并以此提供一个更好的用户体验。

在编写复杂的前端项目时，将会花费大量的时间在编写与 DOM 操作相关的功能。

2. 数据与模型

前端开发中所需要的数据都是通过 API 从后台获取的。我们需要通过 HTTP 请求加载远程数据，或者数据发送到服务器端。而对后台开发人员来说，他只需要从数据库中从获取相应的数据。因此，这样的系统从 MVC 框架上看就类似于图 9-2 所示的形式。

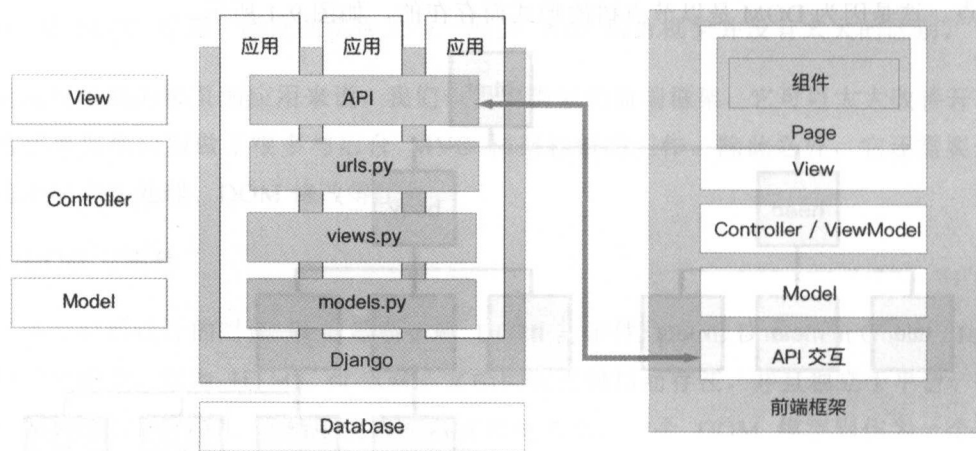


图 9-2 前后台 MVC 框架

我们通过 Ajax 请求（即通过 XHR API）在浏览器的后台向服务器请求数据。当数据从服务器返回后，将根据数据的类型动态修改页面的内容。

在第 5 章中，我们写的博客返回的是一个包含所有博客的对象：

```
def blog_list(request):
    return render_to_response('blog/list.html', {
        'blogs': Blog.objects.all()
    })
```

在这个对象里，我们返回了博客所拥有的一系列属性：标题、作者、链接、内容、发布日期等。当我们为前端设计这个 API 时，需要用一定的格式来返回这些对象，如下是使用 JSON 返回数据：

```
{
    "title": "test",
```

```

"author": {
  "username": "phodal",
  "email": "h@phodal.com"
},
"body": "fdsafafsaf",
"slug": "ffasdfasf",
"id": 1
}

```

过去在后台定义的模型现在又需要重新定义——为前台应用定义模型。两者稍有不同的是，为前台定义的模型应该会基于为后台定义的数据模型。而后台定义的数据模型则应该包含一个对象完整的字段，如当我们创建一个博客的后台模型时，需要定义好所有的字段。而当前台去显示、获取相关的数据时，我们只关注于所需要的字段，如可能不需要博客的 ID。因此，从 API 获取这些数据的时候，只获取所需要的部分。

当开发一个单页面应用的时候，需要为前端创建数据模型的 API，并定义、设计好相应的键值。当第 5 章中，我们的博客列表页只用 80 个字符来显示博客简介，这时就应该在后台处理好，而不是交由前台来处理这些字符。

当然，如果一次返回所有的内容或者某一页内容的数据量不大时，就可以考虑使用一次性返回所有的内容，同时在客户端做一个缓存。这样，当我们访问博客详情页的时候，就不需要再获取一次 API，可以加快用户体验。

在一些复杂的环境下，可能需要获取多次 API 才能得到所有的信息，这就需要考虑一些更好的办法来减少 API 请求。在一些网站上登录的时候，可能需要从多个接口获取数据，这时就需要考虑使用一些技术改进 API 的使用流程，诸如 GraphQL 可以由客户端控制请求的内容。

3. 控制器与状态

当我们成功地获取数据之后，需要相应地修改应用中一些变量的状态、值，再动态显示到页面上，这时就需要由控制器来进行一些处理。举一个比较简单的状态示例：数据的加载状态（Loading，通常会使用动画来表示），即：开始获取数据时，我们会在页面上显

示一个加载的动画，这时可能会用 `loading = true` 来表示正在加载。

- 成功地获取数据后，就会删除或者隐藏这个动画，这时的 `loading` 值就会变为 `false`。
- 获取数据失败后，我们仍旧会将 `loading` 值变为 `false`。与此同时，我们可能会有其他的状态来表示数据加载失败，再让用户重新刷新。

一般来说，一个需要获取数据的控制器都需要这样的状态表示。对一个展示博客的控制器来说，只要有获取数据和状态显示，就可以完成状态的改变。当我们获取数据后，只需要结合模板来渲染页面即可。这一点与编写 Django 中的 HTML 模板是类似的：

```
<h2 class="card-title">
    {{blog.title}}
</h2>
<p>
    {{blog.body}}
</p>
```

当我们执行操作、获取数据后，需要让页面的内容做出相应的变化。这时还需要知道单向及双向数据绑定的概念。

- **单向绑定**，即数据只能从控制器向页面流动，而无法由页面直接向控制器反馈。当我们把数据和模板一起渲染成 HTML 后，要获取 HTML 中相应的值，只能通过事件来触发。
- **双向绑定**，即数据的变更能及时显示到界面上，界面上相应的操作也能变更到模型中。当我们获取新的数据后，HTML 中的内容就会动态地发生变化。与此同时，当我们在页面中输入一些值时，控制器中的模型也会发生变化。

在双向绑定里，只要数据发生了变化，页面就会自动发生变化。对博客详情页来说，展现不了双向绑定的能力，它更适合处理表单，如登录操作。

当我们设计一个表单操作时（诸如注册用户），借助于双向绑定，就可以实时获取到

用户的输入值，并且可以实时做出一些校验，如：输入的用户名中是否带有特殊符号、输入的密码是否达到限制的长度。注册时，当用户输入完用户名，切换到下一个选项，我们就可以获取用户名并通过 API 来查询是否与已有的用户名重复。在这个过程中，用户只需要按照注册流程，并保证自己输入的内容是对的，用户就可以一次注册成功。

在这个过程中里，就是输入值到控制器，并输出值到页面中，由控制器对值进行相应的一些操作，并与服务器进行交互。

4. 路由与页面

与后台路由相似的是，在前端框架中也会有路由的概念。它的原理与后台的 MVC 是类似的：使用对应的函数来处理对应的路由。有所不同的是：前端中的类 MVC 框架的路由变化中，并不会发生实际的 URL 跳转，只会修改相应的 URL——在一些框架里，URL 可能会以 `/#/blog/` 的形式存在，代表这是一个前端的路由，并交由相应的控制器来处理。控制器就可以根据 URL 中的参数执行相应的操作。

同时，在这个过程中会将页面中相应的事件等操作权交由其他控制器来管理。这可以避免不同的控制器控制同一个页面带来混乱。

在 Django 里，我们使用 `urls.py` 来定义路由和控制器之间的关系，也可以使用单一的路由或使用正则表达式。同样，对前端框架来说也是如此，如下是 Angular 2、Vue.js 用于定义路由的示例代码：

```
{ path: 'blog/', component: BlogListComponent },  
{ path: 'blog/:id', component: BlogDetailComponent },
```

由于 Angular 2 与 Vue.js 都将以组件来命名不同的页面，因此其路由看上去是一致的。在上面的代码里，当我们跳转到 `blog/` 页面时将执行 `BlogListComponent` 中相关的函数；访问 `blog/1` 时，将会把 1 作为参数传给 `BlogDetailComponent` 组件来处理。

同样，这种方式对另一个流行的前端框架 React 也是类似的：

```
<Route path="/" component={App}>  
  <IndexRoute component={Home} />
```

```
<Route path='blog' component={BlogList} />
<Route path='blog/:id' component={BlogDetail} />
</Route>
```

它们的表达方式都是相似的，只是形式有所不同而已。

因此，当我们已经有了一个后台应用时，要将其转换为前端应用也是相当容易的一件事。

9.1.2 API 设计与框架选型

随后，摆在我们面前的问题有两个：一个是 API 设计，另一个是框架选型。前者会影响后台系统的流畅程度，后台则会影响系统的开发过程。

1. Web 应用的 API 设计

API 设计是一个相当复杂的问题，不同的业务模式下会有不同的 API 模式。市面上已经有一系列的书在讨论 Web 应用的 API 设计的问题，如 *RESTful Web APIs*、*RESTful Web Services Cookbook* 等。这些书主要讨论的是 RESTful API 的设计，RESTful 即（Representational State Transfer）表述式状态转换，它是由 Roy Thomas Fielding¹ 博士在其博士论文 *Architectural Styles and the Design of Network-based Software Architectures* 中引入的。

RESTful API 讨论的是如何以资源的方式对 API 进行管理。因此，它借助于 HTTP 1.1 提出的四种 HTTP 方法定义了一个规范，这四个方法如表 9-1 所示（除了上面的四种请求类似，HTTP 协议里还有 PATCH、HEAD、OPTIONS 等方法），它对应于数据库中表 9-1 数据的操作。

¹ 他是 HTTP 协议（1.0 版和 1.1 版）的主要设计者、Apache 服务器软件的作者之一、Apache 基金会的第一任主席。

表 9-1

HTTP 方法	执行的操作	数据库操作
GET	获取一个资源	获取 (Retrieve)
POST	创建一个资源	创建 (Create)
PUT	更新一个资源	更新 (Update)
DELETE	删除一个资源	删除 (Delete)

除了上面这些内容，RESTful 还需要使用 URI 定义好资源。使用路径来指定资源，这其中的斜杠则用于划分资源。当我们使用博客来创建资源时，URI 应该可以直观地表述资源，如：

- /blog/，用于返回所有的博客资源。
- /blog/1，用于找到博客 ID 为 1 的资源。

与之前使用不同的 slug 来区分博客所不同的是：最初的设计是出于 URI 的可读性考虑的。而当我们设计 API 时，就需要考虑便利性的问题。除此之外，对资源的操作需与上面的四个操作对应，如表 9-2 所示。

表 9-2

HTTP 方法	资源	解释
GET	/blog/	获取所有的博客
GET	/blog/1	获取某个具体的博客
POST	/blog/	新建一个博客
PUT	/blog/1	更新某个具体的博客
DELETE	/blog/1	删除某个具体的博客

对 POST 和 DELETE 来说，我们都需要使用具体的资源来操作。而这些资源是使用 URI 来指定的，表达资源的方式有两种：资源集合和单个资源。即上面所有的博客是资源集合，某一篇特定的博客则是单个资源。

当为博客添加评论系统后，就可以将 API 设计为/blog/1/comments/，用于获取 ID 为 1 的博客的所有评论；而/blog/1/comments/则对应于 ID 为 1 的博客的第一篇评论。在这个过程中，我们还需要注意 URI 层级过深、过长的 URI 不易于维护。

然而在设计 API 中难的部分并不是这些对资源的操作、设计 API 接口以及对应的 URI，而是去遵循 RESTful 中提出的架构风格的约束设计。

- 客户端。服务器分离，即划分客户端与服务端的职责。客户端不关心服务器端的实现，只关心提供的 API；同样，服务器端也不关心客户端的数据显示等。
- 无状态，服务器不存储客户端请求的上下文信息。每次客户端发出请求时，总是包含完整的信息，服务器端根据这些信息对请求做出完事的响应。
- 可缓存，缓存机制可以提高应用性能，同时改善无状态带来的性能问题。
- 统一接口，使用唯一风格的接口来设计 API。
- 分层设计，用于提高系统各层次的独立性。
- 按需代码（可选），允许对客户端的功能进行扩展。

当设计的服务满足上面的约束时，才称为“RESTful”；反之，如果违反了上述任何一条约束（第六条除外），就不能被称为 RESTful。

除了这些，我们应该让设计的 API 尽量满足 Web 标准，并且应该很容易地在浏览器上进行调试。同时，为了确保 API 安全，最好的方式是使用 HTTPS 进行数据传输，来提高系统的安全性——这需要一些额外的花费及配置，如 HTTPS 证书、配置，并考察它对服务器性能的影响等。

在一些场景下，可能还需要过滤、排序、搜索等一些高级功能。这里将会在博客列表页的 API 里引入分页及超媒体的功能——出于编写更简洁的客户端代码的目的，而引入这些高级的功能。

2. JSON

JSON 如今似乎已经作为前端与后台进行数据交换的标准了，当然，我们还能看到在一些老式应用的 API 中使用 XML 作为交互格式。然而 XML 不仅在解析上不如 JSON 容易——浏览器内建对 JSON 的解析，还需要占据额外的字节。

JSON (JavaScript Object Notation) 是由道格拉斯·克罗克福特构想设计、轻量级的数据交换语言。它的格式完全独立于语言，你可以使用 C++、C#、Java、Python 等来解析 JSON，并且它很容易阅读——基于键值 (Key-Value) 的形式来呈现，如下是一个 API 返回的 JSON 数据。

```
{
  "count": 6,
  "next": null,
  "previous": "http://127.0.0.1:8000/api/blog/",
  "results": [
    {
      "title": "324234234",
      "author": {
        "username": "phodal",
        "email": "h@phodal.com"
      },
      "body": "423423",
      "slug": "23442",
      "id": 6
    }
  ]
}
```

在上面的代码里，第二行的 `count` 作为一个 key，而 6 则是 `count` 的值。它不仅看上去很直观，并且使用 JavaScript 来解析它也相当容易。将上面的结果赋予 `result` 变量，随后就可以从这个 JSON 对象里取出 `count` 的值：`result.count`，或者是以 `result['count']` 的形式——在 JavaScript 里，像这种关联数组就是对象，对象也就是关联数组，因此可以取到这个值。

当我们通过 API 获取到数据时，返回的 API 结果是一个字符串，需要使用浏览器自带的 `JSON.parse()` 方法来将其转换为 JSON 格式。而当我们打算将这些数据存储到本地的缓存时，就需要 `JSON.stringify()` 来将其转换为字符串。

3. 前端选型：单页面应用框架

在今天看来，选择前端框架似乎是一件很难的事，然而这件事情并不是看上去那么难。只是有时候你只想追随潮流，或者因为你在技术选型上受到一些影响。但是总的来说，选择一个框架并不是一件很难的事，同时也不是一件非常重要的事，因为框架本身是相通的。如果我们不尽量去解耦系统，那么选择什么框架都一样。

(1) Angular 2

由于 Angular 2 的推出，在技术选型上没有理由再考虑使用 Angular.js 1.x——因为 Angular 2 的升级方式比较激进，因此，Angular 1.x 的应用几乎需要重写才能在 Angular 2 上运行。值得注意的是，Angular.js 1.x 仍然拥有庞大的生态系统——第三方组件、开源项目、文档、资源，它可以帮助我们快速开发应用。也因此，如果这是一个快速交付的项目，就可以将其列入考虑范围。

Angular 2 即新一代的 Angular，它在性能与渲染上有很大的提升，同时也更容易理解和使用，并且可以使用一套框架在多种平台上构建，即同时适用手机与桌面。Angular 对后端人员写前端代码来说，是一个非常不错的选择，并且最流行的混合应用框架 Ionic 2 也是基于 Angular 2 的。

(2) React

React 是由 Facebook 开发的一款 JavaScript 库，相当流行，并深受开发者欢迎。虽然它只是一个视图层，需要辅以其他框架才能完成更多的工作——搭配 React Router、Redux 就能完成应用的开发。React 在代码层操作虚拟 DOM，渲染时才对真实的 DOM 进行更新，借此来改善应用性能。同时，React 也将组件化实施得很透彻。不过学习 React 就需要学习一系列相关的技术栈，其学习成本相对较高。

与 Angular 2 类似，React 还有一个不错的杀手锏——React Native，虽然这个框架还在有条不紊地挖坑中。但是这带来了一系列的变化，以后只需要一次开发就可以多处运行，再也没有比这更爽的事情发生了。值得注意的是，使用 React Native 要求你拥有原生应用的开发经验，也带来一定的开发成本。

(3) Vue

Vue.js 是一个轻量级的前端框架。它是一个更加灵活开放的解决方案,允许用户以希望的方式组织应用程序,你可以将它嵌入一个现有页面而不一定要做成一个庞大的单页应用。Vue.js 在 API 和设计上比 Angular.js、React 简单,并且更容易上手使用。

同样,它也有一个移动应用的方案 Weex,不过这个方案目前仍然不是很成熟。

除此之外,你还可以考虑使用 Ember.js,它在开发效率上特别显著,拥有强大的命令行功能——只需要执行生成命令,就可以生成相应的组件代码。

9.2 创建移动应用

当我们开发下一代移动 Web 应用时,就需要考虑将混合应用与移动 Web 合一。如果我们不能将它们合并到一起,就应该尽可能地去共用它们的代码。在开发开源应用 Growth 的时候,我尝试使用 Ionic 来构建桌面、Web 及手机移动应用,其构架如图 9-3 所示。

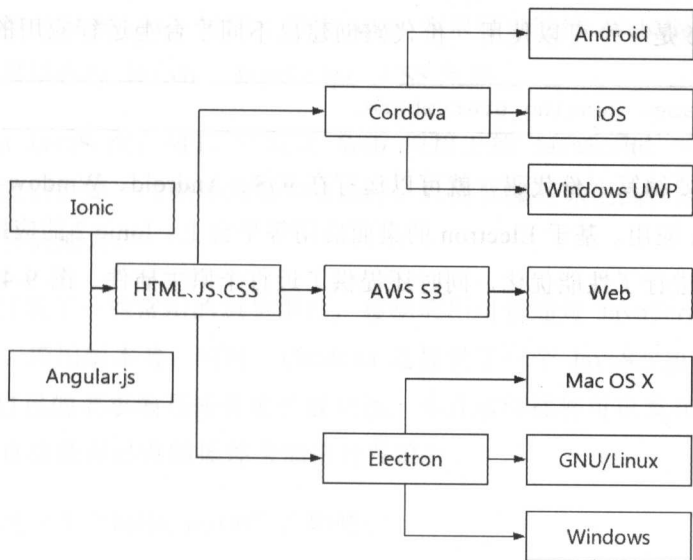


图 9-3 Growth 应用架构

Ionic 和 Angular.js 运行在 WebView 上，因此可以很容易地将其运行在任何带有 WebView 的平台上，如移动应用平台 Cordova、桌面应用平台 Electron，同时它也可以运行在浏览器上。这时需要关注以下两点：

- 响应式设计，面向不同屏幕大小的设备设计出不同的 UI。
- 平台/设备特定代码，在手机上可以调用一些原生插件来提升用户体验，这些代码都应该只在某个平台上才运行。

当有了 Web 应用之后，我们的应用显然没有理由再运行在桌面应用上。然而，当一个移动 Web 应用时，一个移动应用就显得非常有必要。

因此，要引入 Ionic 开发一个移动应用，同时这些也可以将其代码运行在移动网页上。

9.2.1 使用 Ionic 2 创建应用

Ionic 是一个优秀的移动应用开源框架。它提供了一份漂亮的 UI 组件，通过引用这些组件就能方便地创建出漂亮的应用。在其官网显示已经有 3000000 个应用基于 Ionic 构建，并且 Ionic 是一个可以使用一份代码创建出不同平台上运行应用的框架，即：

One code base. Running everywhere.

因此，只需要编写一份代码，就可以运行在 iOS、Android、Windows Phone、Chrome 浏览器上的 PWA 应用、基于 Electron 的桌面应用等平台上。Ionic 2 的应用基于 Cordova，并且有针对性地进行性能优化，同时还提供了近百个原生插件。图 9-4 是基于 Cordova 应用的架构。

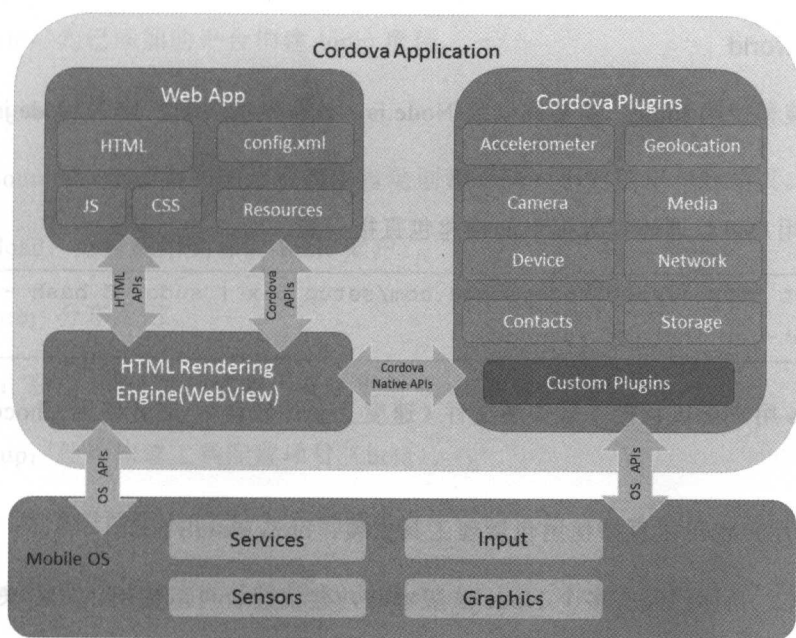


图 9-4 Cordova 应用架构

Cordova 为构建混合应用提供了两个关键性的部分。

- 应用容器（WebView）封装，即让 Web 应用可以通过 WebView 运行在移动设备上，可以执行 HTML、JavaScript、CSS 代码。
- Cordova JavaScript API，可以让 Web 应用上的 JavaScript 与原生平台进行交互。当我们需要一些额外的原生功能时，只需要编写原生插件，应用就可以获得这些组件的能力。

Cordova 还封装了一些常用的原生组件，移动应用可以通过 JavaScript 访问原生设备功能，如摄像头、应用版本等。同时，Cordova 还提供了一个 JavaScript 与原生代码之间的 API，可以按自己的需求编写插件来扩展功能，并且这些插件可以发布到相应的官方仓库，开发者可以直接使用已有的插件来加快开发流程。

下面先从创建一个“hello, world”开始吧。

1. hello,world

为了安装和使用 Ionic，需要先安装 Node.js，按官网的要求，应为 Node.js 6.0 及以上的版本。

Ubuntu 用户可以通过官方提供的构建包直接安装：

```
curl -sL https://deb.nodesource.com/setup_6.x | sudo -E bash -  
sudo apt-get install -y nodejs
```

Windows 用户可以直接下载安装文件（速度上会比较快），或者使用 Chocolatey 安装：
cinst nodejs。

Mac OS 用户也可以直接使用包管理工具安装：brew install node。

除此之外，当你已经有多个 Node.js 版本时，建议读者可以使用 NVM 来管理不同的 Node.js 版本。

接着可以直接安装 Ionic 和 Cordova：

```
npm install -g ionic cordova
```

我们可以直接使用 Cordova 来创建一个项目：cordova create 项目名。但是以这种方式创建的项目还需要搭建构建系统，并整合代码等一系列复杂的步骤。而 Ionic 对 Cordova 中的一些工具进行了封装，并创建出一些更实用的命令。

Ionic 提供了一个命令行工具 ionic，包含一系列的命令可以加速开发混合应用。

- start: 创建一个新的 Ionic 项目。
- serve: 在本地运行应用的开发服务器。
- platform: 为 Ionic 应用添加构建平台。
- run: 在一个已经连接的移动设备上运行 Ionic 项目。
- emulate: 在仿真器/模拟器上运行 Ionic 项目。

- **build**: 为已添加的平台构建 Ionic 项目。
- **plugin**: 添加一个 Cordova 项目。
- **resources**: 从已有的图片资源中自动创建不同大小的图标和启动页。
- **upload**: 上传应用到你的 Ionic 账户。
- **share**: 分享应用。
- **lib**: 获取 Ionic 库的版本, 或者更新 Ionic 的库。
- **setup**: 使用构建工具配置项目 (beta)。
- **io**: 集成应用到 ionic.io 平台的服务。
- **security**: 存储应用的凭证到 Ionic Platform。
- **push**: 上传 APNS 和 GCM 凭证到 Ionic Push。
- **package**: 使用 Ionic Package 来构建应用。
- **config**: 为设置配置变量。
- **browser**: 为平台添加另一个浏览器。
- **service**: 添加一个 Ionic 服务包, 并安装所需要的插件。
- **add**: 为项目添加 bower、Ion 组件以及第三方插件。
- **remove**: 从项目中删除 bower、Ion 组件以及第三方插件。
- **list**: 列出项目中的 bower、Ion 组件以及第三方插件。
- **info**: 列出用户运行环境的所有信息。
- **help**: 提供帮助信息。
- **link**: 为项目创建一个关联的 Ionic App ID。

- **hooks**: 管理 Ionic Cordova 钩子。
- **state**: 使用 `package.json` 为应用存储状态。
- **docs**: 打开 Ionic 文档。
- **generate**: 生成页面及组件。

现在，我们可以用第一个命令 `start` 来创建项目。由于使用 Ionic 2 创建的项目，因此需要加上 `--v2` 参数：

```
ionic start growthStudioApp --v2
```

在这个过程中，它将下载 Ionic 2 应用的基础项目，并执行如下相关的自动配置语句：

```
Creating Ionic app in folder /Users/fdhuang/test/growthStudioApp based on
tabs project
Downloading:
https://github.com/driftyco/ionic2-app-base/archive/master.zip
[=====] 100% 0.0s
Downloading:
https://github.com/driftyco/ionic2-starter-tabs/archive/master.zip
[=====] 100% 0.0s
Installing npm packages...
```

下载完后，将自动安装项目所需要的 NPM 包。安装完成后，我们就可以看到应用成功安装，如图 9-5 所示。

我们可以在应用的目录下运行 Web 服务：

```
cd growthStudioApp
ionic serve
```

```

🎵 🎵 🎵 Your Ionic app is ready to go! 🎵 🎵 🎵

Some helpful tips:

Run your app in the browser (great for initial development):
  ionic serve

Run on a device or simulator:
  ionic run ios[android,browser]

Test and share your app on device with Ionic View:
  http://view.ionic.io

Build better Enterprise apps with expert Ionic support:
  http://ionic.io/enterprise

New! Add push notifications, live app updates, and more with Ionic Cloud!
  https://apps.ionic.io/signup

New to Ionic? Get started here: http://ionicframework.com/docs/v2/getting-started

```

图 9-5 创建应用成功

运行 `ionic serve` 时，ionic 需要执行相应的构建工作，才能启动我们的服务，如：

- 删除旧的构建内容，即 `www/build` 目录下的文件。
- 编译 SASS 到 CSS。
- 编译文件成 HTML。

最后，它将启动一个 Web 服务，其 URL 为 `http://localhost:8100`，整个过程的内容如下：

```

> ionic-hello-world@ ionic:serve /Users/fdhuang/write/growthStudioApp
> ionic-app-scripts serve

```

```

[11:16:13] ionic-app-scripts 0.0.44
[11:16:14] watch started ...
[11:16:14] build dev started ...
[11:16:14] clean started ...
[11:16:14] clean finished in 5 ms
[11:16:14] copy started ...
[11:16:14] transpile started ...

```

```
[11:16:18] transpile finished in 3.62 s
[11:16:18] webpack started ...
[11:16:18] copy finished in 4.01 s
[11:16:23] webpack finished in 4.89 s
[11:16:23] sass started ...
[11:16:24] sass finished in 1.20 s
[11:16:24] build dev finished in 9.73 s
[11:16:24] watch ready in 9.87 s
[11:16:24] dev server running: http://localhost:8100/
```

接着，打开上面的 URL：<http://localhost:8100/>，就可以看到应用。Ionic 2 除了可以运行在不同的平台外，它还为不同的平台创建了相应的 UI。图 9-6 是使用 `ionic serve --lab` 命令运行的服务，可以看到三种不同的操作系统下的界面。

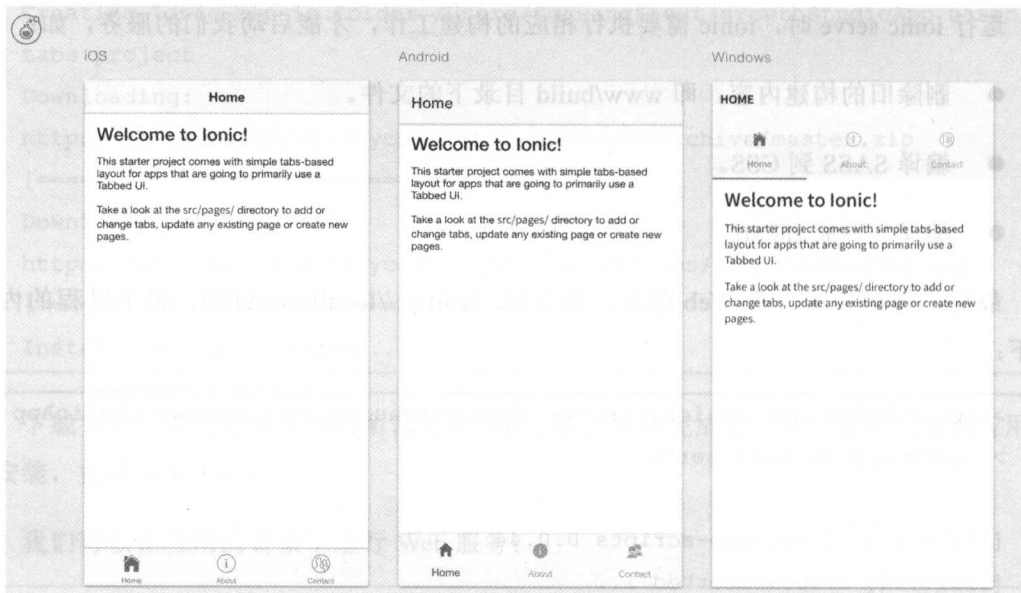


图 9-6 Ionic Web 预览界面

图 9-6 中的三个操作系统分别为 iOS、Android、Windows Phone，它们在 UI 风格上都各有不同，并且这些 UI 的设计也符合各自操作系统上推荐的一些规范。这也是为什么选择 Ionic 的重要原因。

2. 应用目录概览

下面让我们到 `growthStudioApp` 目录下了解应用的大致结构。

```
|— config.xml
|— ionic.config.json
|— package.json
|— src
|— plugins
|— platforms
|— node_modules
|— resources
|— hooks
|— www
```

其中：

- `config.xml`，Cordova 的全局配置文件，Cordova 将会根据这个配置文件生成应用。
- `plugins`，用于放置应用的插件。
- `hooks`，放弃钩子脚本——在构建的时候，依据不同的钩子类似来执行一些相应的操作。
- `ionic.config.json`，ionic 的一些相关配置选项。
- `package.json`，存放相应的 node.js 的包依赖。
- `www`，用于存放最后构建出来的内容，以及一些静态资源。
- `src`，用于存放编写的源码。
- `platforms`，存放生成的不同平台的代码，如 iOS、Android 项目的代码。这些代码可以直接使用相应的 IDE 来编译及运行。

在 `config.xml` 中，我们会指定应用的入口文件，即下列代码中的 `content` 标签。

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<widget id="com.ionicframework.growthstudioapp906019" version="0.0.1"
xmlns="http://www.w3.org/ns/widgets"
xmlns:cdv="http://cordova.apache.org/ns/1.0">
  <name>growthStudioApp</name>
  <description>An awesome Ionic/Cordova app.</description>
  <author email="hi@ionicframework"
href="http://ionicframework.com/">Ionic Framework Team</author>
  <content src="index.html"/>
  ...
</widget>
```

上面中的 `widget` 标签包含了应用的相关配置：`id` 即应用的包名，`version` 则对应应用的版本。同时，`name` 标签则用于显示在移动设备上的应用名，`description` 标签则对应应用的概述。除此之外，还有作者信息、相应权限的配置等。在为应用创建了图标和启动页后，相应的配置也会写在这个文件里。

我们编写的源码位于 `src` 目录下：

```
├─ app
│   ├── app.component.ts
│   ├── app.html
│   ├── app.module.ts
│   ├── app.scss
│   ├── main.dev.ts
│   └── main.prod.ts
├─ pages
│   ├── home
│   │   ├── home.html
│   │   ├── home.scss
│   │   └── home.ts
│   └── tabs
│       └── tabs.html
```

```

|       └─ tabs.ts
|
├─ assets
├─ declarations.d.ts
├─ index.html
├─ manifest.json
└─ theme

```

在目录 `index.html` 下使用 `config.xml` 指定的入口。这个文件和之前的着陆页没有太大的区别，只需要在其中引入 JavaScript、CSS 文件。不过，在这里会先使用 TypeScript² 来编写代码，在运行时会将其转化为 JavaScript 代码。同时对 CSS 文件也是类似的，在编程时会使用 SCSS³，而在构建和运行时会将其转换为 CSS 文件。即在编写的时候使用 TypeScript 和 SCSS，但是在运行前将其转换为 JavaScript 和 CSS。

让我们再回到上面的目录。

- `app` 目录，包含应用的初始化代码、组件代码、不同环境的配置等。
- `pages` 目录，我们将页面都放置在这个目录下，每个页面都包含了独立的 `html`、`ts`、`scss` 文件。其中的 `tabs` 文件是应用的标签栏（Tabbar），它也是各个主页面的入口。
- `assets` 目录，存放资源文件。
- `declarations.d.ts`，声明文件，让 TypeScript 编译器知道一个对象的类型信息。
- `manifest.json`，存储 Google 推出的 Progressive Web Apps 应用的相关配置。
- `theme` 目录，放置一些与全局主题相关的配置。

下面先看看首页的代码。

2 TypeScript 是一种由微软开发的自由和开源的编程语言。它是 JavaScript 的一个超集，而且本质上向这个语言添加了可选的静态类型和基于类的面向对象编程。

3 SCSS 是一种预处理器，它与 CSS 兼容，并且扩展了 CSS 的语法，添加了变量、嵌套、混入等功能。

3. 应用原型与页面组成

理想情况下，我们应该和第 2 章一样：使用一些原型工具来设计出 App 的原型。限于篇幅原因，就不详细介绍如何使用原型工具来创建原型了，其过程和第 2 章类似。不过，推荐读者可以使用 Adobe Experience Design 来设计移动应用，它可以为页面中的元素创建链接到新的页面，使其表现起来和真实的应用类似。

为 App 设计以下功能。

- 首页：将介绍如何使用 Ionic 的轮播模块和卡片模块。
- 博客列表与详情页：将介绍如何结合 API 获取数据。
- 登录页：将介绍用户授权的机制。

当前的移动应用在设计上和 Web 页面类似，一般都由三部分组成：Header、Content 和 Footer。Header 即对应页面的标题，而 Content 则是页面的内容，Footer 是底层的标签栏。在一些应用上可能会有一些细微的差异，总体上还是相似的。

在生成的首页模板文件 home.html 里，可以看到下面的代码：

```
<ion-header>
  <ion-navbar>
    <ion-title>Home</ion-title>
  </ion-navbar>
</ion-header>

<ion-content padding>
  <h2>Welcome to Ionic!</h2>
  <p>
    This starter project comes with simple tabs-based layout for apps
    that are going to primarily use a Tabbed UI.
  </p>
  <p>
    Take a look at the <code>src/pages/</code> directory to add or change tabs,
```

```
update any existing page or create new pages.
```

```
</p>
```

```
</ion-content>
```

上面的代码分成以下两部分。

- **ion-header**: 即应用顶部的内容, 通常会包含当前页面的标题、返回按钮等内容。
- **ion-content**: 即页面的内容, 主要就是编写这部分内容。

我们在页面中看到的另一部分代码则会定义在 `tabs.html` 中。这是因为这部分内容是共用的, 因此, 它被提取为一个独立的组件。

```
<ion-tabs>
```

```
  <ion-tab [root]="tab1Root" tabTitle="Home" tabIcon="home"></ion-tab>
```

```
  <ion-tab [root]="tab2Root" tabTitle="About" tabIcon="information-circle">
```

```
</ion-tab>
```

```
  <ion-tab [root]="tab3Root" tabTitle="Contact" tabIcon="contacts">
```

```
</ion-tab>
```

```
</ion-tabs>
```

我们在 `tabs.html` 中定义标签栏的图标和文字, 以及其对应的页面。在 `tabs.ts` 文件中指定了不同标签的根目录与页面的关系:

```
export class TabsPage { tab1Root: any = HomePage; tab2Root: any = AboutPage;
tab3Root: any = ContactPage; ... }
```

当我们在应用上单击相应的标签时, 就会跳转到相应的页面。在上面看到的是一个普通用户看到的应用。

4. Ionic 2 应用初始化过程

由于这部分涉及初始化过程的内容比较复杂, 建议经验较少的读者可以先跳过这部分内容。

对开发人员来说, `app.module.ts` 文件才是应用的真正入口 (也称为根模块)。我们需

要在文件中添加所有的页面，内容如下：

```
import { NgModule } from '@angular/core';
import { IonicApp, IonicModule } from 'ionic-angular';
import { MyApp } from './app.component';
...

@NgModule({
  declarations: [
    MyApp,
    AboutPage,
    ContactPage,
    HomePage,
    TabsPage
  ],
  imports: [
    IonicModule.forRoot(MyApp)
  ],
  bootstrap: [IonicApp],
  entryComponents: [
    MyApp,
    AboutPage,
    ContactPage,
    HomePage,
    TabsPage
  ],
  providers: []
})
export class AppModule {}
```

在上面的代码里，向@NgModule 中传递了一系列的元数据，这些数据将帮助 Angular 2 编译器来更好地处理模块。NgModule 是一个装饰器函数，它接收一个用来描述模块属性的元数据对象。每个 Angular 应用至少需要含有一个模块类——根模块（通常叫作

AppComponent)，并且每个模块都带有一个@NgModule 装饰器。我们传入了一系列的属性。

- **declarations**，可声明类的列表，让模块中的其他指令可以使用，包含这个模块中的组件（Component）、指令（Directives）、管道（Pipes）。
- **imports**，用于引入应用需要的模块特性，引入的文件都是 NgModule 的类。默认的 Angular 2 应用将引入 BrowserModule 模块，而 Ionic 应用引入的则是 IonicModule.forRoot(MyApp)。
- **bootstrap**，指定应用的主视图（或称为根组件），这里默认使用的是 IonicApp，默认会自动加入 entryComponents 数组。
- **providers**，依赖注入提供商（providers）的列表——将需要的服务引入到这个全局列表中，以便在程序的其他部分使用。
- **entryComponents**，入口组件，Angular 2 会自动将使用到的组件添加到这个数组里。

在我们的应用运行后，将进入 bootstrap 指定的根组件。同时在 imports 中调用 IonicModule.forRoot(MyApp)，IonicModule 是一个 NgModule，用于帮助启动整个 Ionic 应用。它会将传入的 MyApp 组件设置为应用的根组件，并确保框架所有的组件、指令都被提供，因此，程序随后运行到 MyApp 类，即 app.component.ts 文件：

```
@Component({
  template: `<ion-nav [root]="rootPage"></ion-nav>`
})
export class MyApp {
  rootPage = TabsPage;

  constructor(platform: Platform) {
    platform.ready().then(() => {
      StatusBar.styleDefault();
      SplashScreen.hide();
    });
  }
}
```

```

    }

```

我们在模板里指定了 `TabPage` 作为根页面，即 `rootPage = TabPage;`，同时在模板里传入了 `rootPage` 变量，即 `[root]="rootPage"`。Angular 2 的模板可以直接传入字符串，也可以直接传入模板的地址。在构造函数 `constructor` 里初始化应用。`constructor` 会在创建对象 `MyApp` 时自动初始化及执行。因此，将执行这个函数中的语句。

随后在 `TabPage` 中定义不同的根页面：

```

@Component({
  templateUrl: 'tabs.html'
})
export class TabPage {
  tab1Root: any = HomePage;
  tab2Root: any = AboutPage;
  tab3Root: any = ContactPage;

  constructor() {

  }
}

```

由于没有指定根页面，因此将调用代码中指定的第一个页面，即 `HomePage`。这部分逻辑将由 `tabs.html` 模板中使用的 `ion-tabs` 组件来负责。最后程序将运行 `home.ts` 文件，并载入所需要的模板。

```

...
@Component({
  selector: 'page-home',
  templateUrl: 'home.html'
})
export class HomePage {
  ...
}

```

在@Component装饰器中,指定了 selector 和模板的 URL,程序在运行时会载入对应的模板文件。而在其中的 selector 则用来定义外部的模块如何使用这个模块,即我们只需要在另一个模板中引入<page-home></page-home>,就可以引用整个首页。

9.2.2 更新首页

在 Web 版首页里已经拥有一个内容滑动的效果。因此,可以先在移动端实现一个同样的功能,同时添加一些简单的文字,如图 9-7 所示。



图 9-7 首页布局

图 9-7 中的内容区域分为两部分，一部分是灰色区域的区域滑动（Slide），另一部分是一些文字简介。

Ionic 2 提供了一个 Slides 组件，可以让我们轻松地实现这个功能。我们只需要在 home.html 中引用 ion-slides 和 ion-slide 组件即可。

```
<ion-slides>
  <ion-slide>
    <h2>《Growth: 全栈增长工程师指南》</h2>
    <p>
      可以帮助构建你的知识体系，这也是其他技术书籍所欠缺的。它可以告诉你，你可以学习什么，然后看什么书。
    </p>
  </ion-slide>
  <ion-slide>
    <h2>《Growth: 全栈增长工程师实战》</h2>
    <p>在 Growth 中介绍的只是一系列的实践，而 Growth 实战则会带领读者履行这些实践。</p>
  </ion-slide>
  <ion-slide>
    <h2>《Growth: 增长工程师修炼之道》</h2>
    <p>Coming soon...</p>
  </ion-slide>
</ion-slides>
```

ion-slides 标签与 ion-slide 标签的关系类似于 ul 标签和 li 标签——每个滑动的内容都放在 ion-slide 标签内，而所有的 ion-slide 都应该放在 ion-slides 标签中。

接着还需要在 home.scss 中设置相应的高度和颜色。

```
page-home {
  ion-slides {
    height: 16em;

    ion-slide {
```

```
background-color: #777;
padding: 1em;
color: #fff;
}
```

我们将 ion-slides 区域的高度设置为 20 em，同时设置了 ion-slide 的背影颜色、字体颜色，以及边距。为了让内容能自动滑动，可以向 ion-slides 组件传入参数对其进行配置。在下列模板文件中：

```
<ion-slides [options]="mySlideOptions">
```

这个参数需要在 home.ts 中使用相应的参数进行配置，表 9-3 是当前 Ionic Slide 所支持的配置。

表 9-3

属性	值类型	默认	描述
autoplay	number	-	切换 slide 时的动画时间（毫秒为单位）
direction	string	'horizontal'	滑动的方向：'horizontal' 或者 'vertical'
initialSlide	number	0	设置初始 slide
Loop	boolean	false	是否从最后一个到第一个间循环
pager	boolean	false	是否用点来显示当前处于第几个 slide
speed	number	300	在 slides 间切换的时间间隔（毫秒为单位）

因此，我们将配置设置为：

```
export class HomePage {
  public mySlideOptions = {
    pager: true,
    speed: 7000,
    autoplay: 1000,
    loop: true
  };
}
```

```
...
}
```

这样就完成了 Slide 部分，而要完成剩下的部分就比较简单，可以自己布局，或者直接使用 Ionic 提交的一些组件来完成，如 Card。卡片是显示重要内容片段的一种好方式，并且已经成为应用程序设计中的核心设计模式。你可以在很多地方看到它的应用，如微信的朋友圈、首页的不同会话等，这些都是以卡片的方式存在，而卡片主要由两部分组成：card-header 和 card-content。如下是创建的一个卡片式布局：

```
<ion-card>
  <ion-card-header>
    Growth Ebook
  </ion-card-header>
  <ion-card-content>
    Growth 电子书系列可以帮助你更好地构建 Web 开发知识体系。
  </ion-card-content>
</ion-card>
```

在 header 里设置了标题，在 content 里设置了内容。在一些更复杂的设计里，我们会使用一些自定义的样式来完成设计。当我们需要在卡片后有一个相应的操作时，就可以在 ion-card-content 标签后添加一些相应的内容，如：

```
...
</ion-card-content>
<ion-item>
  <span item-right>详细</span>
  <ion-icon name='more' item-right style="color: #d03e84"></ion-icon>
</ion-item>
</ion-card>
```

其效果如图 9-8 所示。

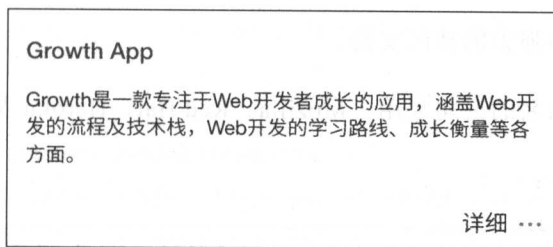


图 9-8 带操作的卡片布局

这些都是关于 UI 设计及组件的使用，读者可以从 Ionic 的官方文档上获取更详细的资料。

9.3 实现博客应用开发

在 9.2 节对 Angular 2 和 Ionic 2 的使用进行了详细介绍，并编写了相应的代码对首页进行完善。接下来，让我们结合已经创建的后台来搭建应用。不过，在那之前需要先创建一个后台 API 来提供给前台使用。

9.3.1 创建博客 API

1. Django REST Framework

为了结合现有的应用创建后台的 API，我们需要用到一个名为 Django REST Framework 的 RESTful API 库。Django REST Framework 这个名字比较直白，就是基于 Django 的 REST 框架，使用该框架的理由如下。

- 可以在浏览器中调试的 API。
- 包括 OAuth1a 和 OAuth2 的认证策略。
- 支持 ORM 和非 ORM 数据资源的序列化。
- 全程自定义开发。

- 额外的文档和强大的社区支持。
- 已经被多家知名的公司采用：Mozilla、Red Hat、Heroku 及 Eventbrite。

同样是安装这个库：

```
pip install djangorestframework
```

然后把它添加到 `dev.txt` 和 `pro.txt` 文件中，以及 `INSTALLED_APPS` 中：

```
INSTALLED_APPS = (  
    ...  
    'rest_framework',  
)
```

接下来，就可以创建我们的 API。

2. 创建博客列表 API

Django REST Framework 可以直接对 ORM 数据源进行序列化，对应这里的博客模型。只需要定义好相应的显示字段，并配置好权限即可。参考 Django REST Framework 的官方文档，可以很快创建出下面的 API 代码。因为需要遵循 Django 应用的分离原则，创建这个 `api.py` 文件，并放置在 `blog` 目录下：

```
from django.contrib.auth.models import User  
from rest_framework import permissions  
from rest_framework import serializers, viewsets  
from blog.models import Blog
```

```
class BlogSerializer(serializers.ModelSerializer):  
    author = User  
  
    class Meta:  
        model = Blog
```

```
fields = ('title', 'author', 'body', 'slug', 'id')
```

```
class BlogSet(viewsets.ModelViewSet):
    permission_classes = (permissions.IsAuthenticatedOrReadOnly,)
    serializer_class = BlogSerializer
    queryset = Blog.objects.all()
```

在上面这个例子中，API 代码主要由两部分组成。

- **ViewSet**，用于定义视图的展现形式，如返回哪些内容，需要做哪些权限处理。
- **Serializers**，用于定义 API 的表现形式，如返回哪些字段，返回什么格式。

在 URL 中会定义相应的规则到 **ViewSet**，而 **ViewSet** 则通过 **serializer_class** 找到对应的序列化类（即转换 ORM 模型），将 **Blog** 的所有对象赋予 **queryset**，并在 API 中返回这些值。同时使用 **IsAuthenticatedOrReadOnly** 设置这个 API 的权限，当用户没有授权时只能读取数据，不能创建新的数据。

在 **BlogSerializer** 中，我们将 **author** 与 **User** 模型进行了关联，用于选择相应的用户作为作者，并在其元数据（即，**class Meta:** 部分）中定义要返回的字段：**title**、**author**、**body**、**slug**、**id**。

接着，在 **urls.py** 中配置相应的 API 的 URL：

```
...

from rest_framework import routers
from blog.api import BlogSet

apiRouter = routers.DefaultRouter()
apiRouter.register(r'blog', BlogSet)

urlpatterns = patterns('',
```

```
...
url(r'^api/', include(apiRouter.urls)),
)
```

我们使用默认的 Router 来配置 URL，即 `DefaultRouter`，它提供了一个非常简单的机制来自动检测 URL 规则。因此，我们只需要注册好 url——`blog`，以及其值 `BlogSet` 即可，随后再为其定义一个根 URL。

3. 测试 API

现在，我们可以从 `http://127.0.0.1:8000/api/` 中访问现在的 API。由于 Django REST Framework 提供了一个 UI 机制，因此可以在网页上直接看到所有的 API，如图 9-9 所示。

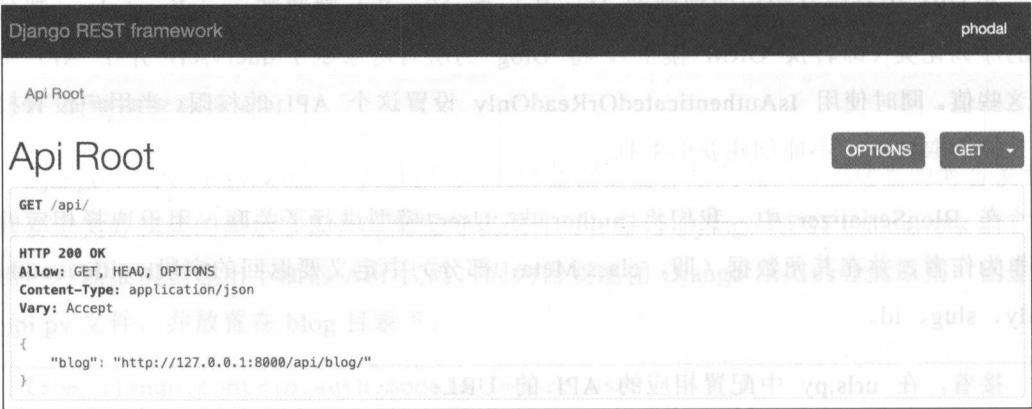


图 9-9 Django REST Framework 列表

然后，单击页面中的 `http://127.0.0.1:8000/api/blog/`，就可以访问博客相关的 API，如图 9-10 所示。

在页面上显示了所有的博客内容，页面下方有一个表单可以用于创建数据，如图 9-11 所示。

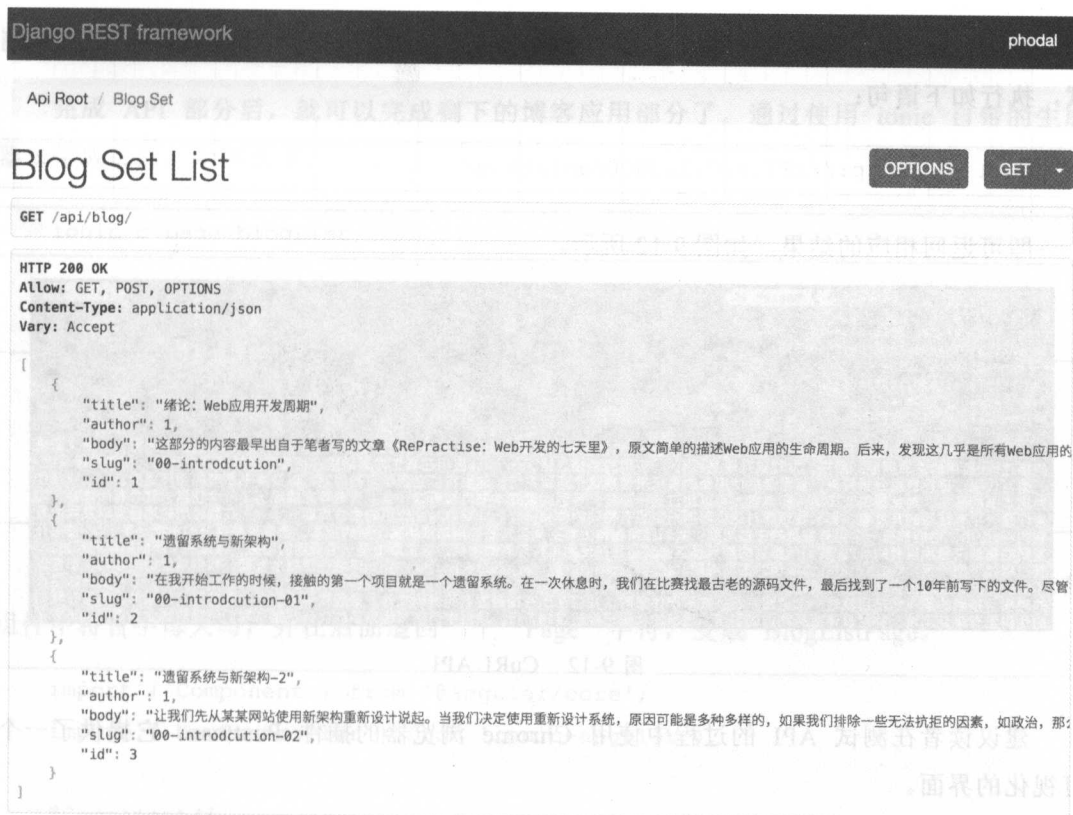


图 9-10 博客 API

Raw data HTML form

标题

博客的标题

作者

正文

URL

POST

图 9-11 创建博客的表单

直接在表单中添加数据，即可完成数据的创建。当然，也可以直接用命令行工具来测试，执行如下语句：

```
curl -i http://127.0.0.1:8000/api/blog/
```

即可返回相应的结果，如图 9-12 所示。

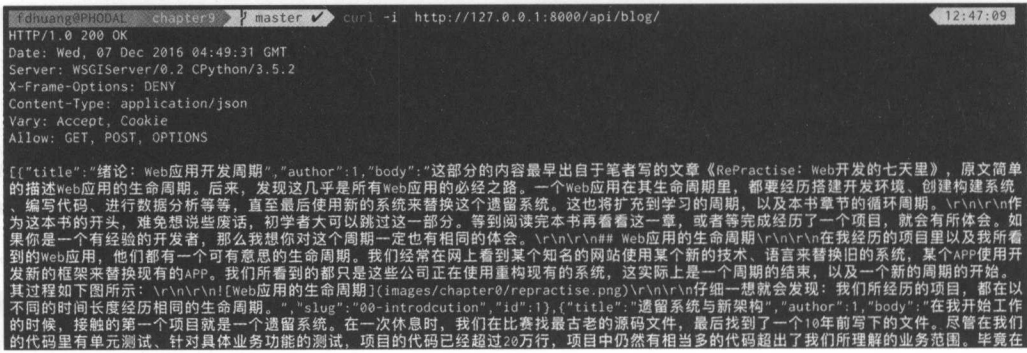


图 9-12 CuRL API

建议读者在测试 API 的过程中使用 Chrome 浏览器的插件 Postman，它提供了一个可视化的界面。

9.3.2 创建详情页和列表页

当我们拥有博客的 API 后，要创建对应的详情页和列表页就变得相当容易，只需要以下操作。

- ①生成详情页与列表页。
- ②编写相应的模板。
- ③从 API 中获取相应的结果。

编写模板作为第二步的主要原因是，可以先在代码中写一些假的数据，随后用真实的数据来替换。这样做可以让拆分步骤实现小步前进。

1. 生成页面

完成 API 部分后, 就可以完成剩下的博客应用部分了。通过使用 `ionic` 自带的生成器来生成页面, 示例如下:

```
ionic g page blogList
```

它将会在 `src/app` 目录下生成 `blog-list` 文件夹。

```
.
├─ blog-list.html
├─ blog-list.scss
└─ blog-list.ts
```

`ionic` 命令会将 `blogList` 风格的驼峰性转换为 `blog-list` 连字号形式的风格。不过在组件中将首字母大写, 并在后面追回一个 "Page" 字符, 变成 `BlogListPage`。

```
import { Component } from '@angular/core';
import { NavController } from 'ionic-angular';
```

```
@Component({
  selector: 'page-blog-list',
  templateUrl: 'blog-list.html'
})
```

```
export class BlogListPage {
```

```
  constructor(public navCtrl: NavController) {}
```

```
  ionViewDidLoad() {
```

```
    console.log('Hello BlogListPage Page');
```

```
  }
```

```
}
```

以及一个空白的模板页面 `blog-list.html`:

```
<ion-header>
  <ion-navbar>
    <ion-title>blogList</ion-title>
  </ion-navbar>
</ion-header>

<ion-content padding>
</ion-content>
```

同样，这个步骤也适用于创建详情页。

2. 创建列表页

创建列表页的过程与之前创建桌面版本的列表页相似。

①显示博客数据。

②添加相应的处理函数来处理从列表页向详情页跳转。

桌面版本从列表页跳转到详情页使用的是 `URL`。而在移动页面则有些不一样，需要先在 `blog-list.ts` 中编写一些数据，把它的值赋予 `blogs`，就可以使用这个值来实现页面及模板。

```
public blogs = [{
  "title": "绪论: Web 应用开发周期",
  "author": 1,
  "body": "这部分内容最早出自笔者写的文章《RePractise: Web 开发的七天里》。",
  "slug": "00-introdcution",
  "id": 1
}, {
  "title": "遗留系统与新架构",
  "author": 1,
  "body": "在我开始工作的时候，接触的第一个项目就是一个遗留系统。",
```

```

"slug": "00-introdcution-01",
"id": 2
  });

```

而移动应用的模板和之前的模板差不多：都是在一个 for 循环里取出一个个博客。之前在 Django 里使用的是 `{% for blog in blogs %}`，而在 Angular 中使用 `ngFor` 指令来遍历一个博客，然后取出其中的值：

```

<ion-content>
  <ion-card *ngFor="let blog of blogs">
    <ion-card-content>
      <h2 class="card-title">
        {{blog.title}}
      </h2>
      <p>
        {{blog.body}}
      </p>
    </ion-card-content>
  </ion-card>
</ion-content>

```

这里同样使用 Ionic 的卡片组件来显示博客。接着，让我们从 API 中获取博客的内容。为了发送一个 HTTP get 请求，我们需要引入 Angular 2 中的 HTTP 模块。然后创建一个 HTTP 请求，语句如下：

```

import {Http} from "@angular/http";
import 'rxjs/add/operator/map';
...

constructor(public navCtrl: NavController, public http: Http) {
  let url = 'http://localhost:8000/api/blog/';
  this.http.get(url)
    .map(res => res.json())
    .subscribe(data => {

```



```
    this.blogs = data;
  });
}
```

我们在构建函数里创建了一个 `http` 对象，然后使用 `this.http.get` 请求 API。Angular 2 引入了流式框架 RxJS⁴，因此，当我们调用 `get` 方法后，便得到一个 `observable`，用于被监听的对象。然后使用 `map` 值将获取到的 API 结果进行格式化，即将字符串转换成 JSON 对象。最后将使用 `subscribe` 方法来订阅返回的结果 `data`，再将这个值赋予 `blogs` 变量。

将代码写入 `blog-list.ts` 文件中，运行后就会发现报错，如图 9-13 所示。

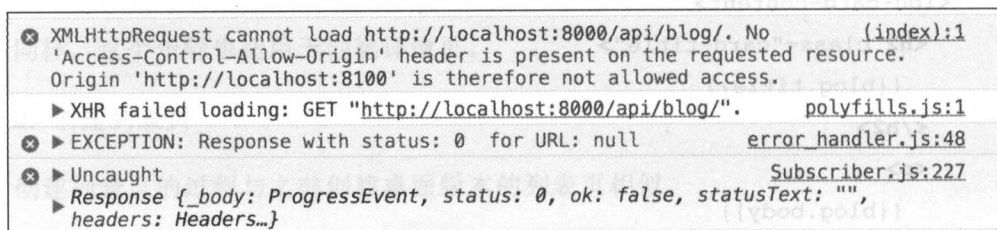


图 9-13 跨域问题

在错误的内容中有一个关键字很重要：'Access-Control-Allow-Origin'，即我们的 API 出于安全原因不允许跨域⁵调用。应对这种问题一般有两种做法。

- 在本地设置代理来允许这种跨域请求。
- 在服务器端允许跨域请求。

从理论上说，在本地设置代理比较安全，但实际上相差不了太多——博客 API 本身是开放的。我们可以设置代理来访问，其他人也可以如此。因此，这里将介绍配置服务器端来允许跨域，并设置一些相应的安全措施。

⁴ RxJS 是使用可观察序列和 LINQ 风格查询操作符来编写异步和基于事件程序的类库。简单地说，它可以帮助我们更好地处理异步数据流。

⁵ 当一个资源请求一个其他域名的资源时会发起一个跨域 HTTP 请求。

3. 添加跨域支持

在 Django 框架里,有一个名为 `django-cors-headers` 的插件用于实现对跨域请求的支持。我们只需要安装它,并进行一些简单的配置即可。

```
pip install django-cors-headers
```

还需要添加 `django-cors-headers=1.2.0` 到 `dev.txt` 和 `prod.txt` 文件中,并添加到 `settings.py` 中。

```
INSTALLED_APPS = (  
    ...  
    'corsheaders',  
    ...  
)
```

接着在中间件中添加这个中间伯。

```
MIDDLEWARE_CLASSES = (  
    'corsheaders.middleware.CorsMiddleware',  
    'django.middleware.common.CommonMiddleware',  
    ...  
)
```

同时,在配置文件中添加相应的白名单 IP 或域名即可。

```
CORS_ORIGIN_WHITELIST = (  
    'studio.growth.ren',  
    'localhost:8100',  
    '127.0.0.1:8100'  
)
```

其中的 `localhost:8100` 和 `127.0.0.1:8100` 是本地开发环境使用的,我们应该将其添加到 `local_settings.py` 文件中,并且在产品环境中不需要这两个变量,甚至也不需要允许跨域请求。这是因为当我们的应用作为一个移动应用运行的时候,它发出的请求就不会遇到这个

跨域问题。

等待 API 重启之后，就会发现移动应用上已经显示了正确的博客内容。

4. 创建详情页

在创建详情页之前，需要先在列表页里创建一个跳转到详情页的入口。而在创建入口之前，需要先生成详情页。

```
ionic g page blogDetail
```

这样在 `blog-list.ts` 文本中才能引入这个模块。然后使用 `navCtrl` 来负责页面间的跳转。

```
gotoBlogDetail(blogId) {
  this.navCtrl.push(BlogDetailPage, {
    blogId: blogId
  });
}
```

我们的 `navCtrl` 是在构造函数里声明的，而 `NavController` 则是用于导航控制器组件的基本类。我们用 `navCtrl` 的 `push` 方法进入新的页面，用 `pop` 方法则可以回到堆栈的下一个页面。在代码里，将 `BlogDetailPage` 组件推入堆栈，将博客的 ID 转入到详情页的组件里，并导航到这个页面。对应的，我们需要在列表页的模板中调用这个方法，并传入博客的 ID。

```
...
<ion-card *ngFor="let blog of blogs" (click)="gotoBlogDetail(blog.id)">
...
```

在 Angular 2 里，以 “()” 开始的属性表示它是一个事件，以 “[]” 开始的属性表示它是一个属性。因此，这里创建了一个 `click` 事件，即当用户点击页面的时候，将执行跳转函数。函数将执行 `blog-detail.ts` 文件中的 `BlogDetailPage` 类。

```
export class BlogDetailPage {
  public blog;
```

```

constructor(public navCtrl: NavController, public http: Http, public
navParams: NavParams) {
  let blogId = navParams["data"]["blogId"];
  let url = 'http://localhost:8000/api/blog/' + blogId + '/';
  this.http.get(url)
    .map(res => res.json())
    .subscribe(data => {
      this.blog = data;
    });
}
}

```

这时将读取传过来的博客 ID，传递到详情页的参数都存储在 `navParams` 变量中。随后和列表页一样：请求博客 API，只是 URL 变成了加上某个特定 ID 的博客。

这部分逻辑和之前编写桌面版代码极其类似，由 Django 来处理对应的博客路由：

```
url(r'^blog/(?P<slug>[^\.\.]+).html', blog_detail, name='blog_view'),
```

再获取 URL 中的参数给 `blog_detail` 方法，再从数据库查询相应的数据，最后显示到页面上。

9.4 用户登录与博客创建

前面介绍的博客部分由于都是 GET 请求，实现起来相对比较容易。而当我们创建一个博客的时候，就需要更复杂的步骤：需要先设计一个登录页面和授权机制，然后才能支撑起创建博客的逻辑。

9.4.1 使用 JWT 实现登录

1. JSON Web Token

与之前使用 Django Admin 来登录并创建有所不同的是：API 不应该存在状态。

当我们在桌面版本登录成功时，会在服务器端生成一条记录，用于表示当前的登录用户，这个记录就是 Cookie。每当我们进行一些操作的时候，服务器端都会要求请求中带有这个 Cookie。同时在这个过程里，服务器端则需要维护一个名为 Session 的列表，用于区分用户。

在这种机制下，我们的服务器都需要维护这些 Session 信息，并且它的设计并不符合 RESTful 的规范。因此，需要一些更好的授权机制：JWT 是为了在网络应用环境间传递声明而执行的一种基于 JSON 的开放标准，其运行机制如下。

- 登录时，客户端向服务器端发送用户名和密码。
- 服务器端验证用户名和密码是否匹配，成功时向客户端发送一个 Token。
- 服务器端需要存储这个 Token，并在以后需要授权的操作中带上这个 Token。
- 服务器端再验证这个 Token 是否有效，随后再执行相应的操作。

为了实现这部分功能，我们仍然可以使用其他框架来完成基础功能。这里用到了一个名为 `django-rest-framework-jwt` 的库，从它的名字可以知道，它就是基于 Django REST Framework 之上的 JWT 实现。还是继续使用 `pip` 来安装这个库，记得把它添加到 `requirements.txt` 中。

```
pip install django-rest-framework-jwt
```

接着在 URL 中配置用于获取 token 的 API:

```
urlpatterns = patterns(
    '',
    # ...
```

```
url(r'^api-token-auth/',
'rest_framework_jwt.views.obtain_jwt_token'),
)
```

同时在 `settings.py` 文件中添加相应的授权配置：

```
REST_FRAMEWORK = {
    'DEFAULT_PERMISSION_CLASSES': (
        'rest_framework.permissions.IsAuthenticatedOrReadOnly',
    ),
    'DEFAULT_AUTHENTICATION_CLASSES': (
        'rest_framework.authentication.SessionAuthentication',
        'rest_framework.authentication.BasicAuthentication',
        'rest_framework_jwt.authentication.JSONWebTokenAuthentication',
    ),
}
```

Django REST Framework JWT 默认设置的过期时间是 300 s，我们可以在设置中将这个时间调长：

```
JWT_AUTH = {
    'JWT_EXPIRATION_DELTA': datetime.timedelta(days=1),
}
```

除此之外，我们还能做以下操作。

- 设置 JWT 处理函数：JWT_PAYLOAD_HANDLER、JWT_PAYLOAD_GET_USER_ID_HANDLER、JWT_RESPONSE_PAYLOAD_HANDLER、JWT_DECODE_HANDLER 和 JWT_ENCODE_HANDLER。
- 做相应的 JWT 配置：JWT_PUBLIC_KEY、JWT_PRIVATE_KEY、JWT_ALGORITHM、JWT_REFRESH_EXPIRATION_DELTA 等。

详细的内容可以参考官方提交的文档。完成上面的步骤之后，我们就可以用 `curl` 命令

或者 Chrome 浏览器的 Postman 来做测试。

- 向服务器发送用户名和密码，获取对应的 Token。

如下是 curl 创建的请求，在其中发送了用户名和密码。

```
curl -H "Content-Type: application/json" -X POST -d '{"username":"root",
"password":"admin"}' http://localhost:8000/api-token-auth/
```

然后服务器端返回了对应的 Token，它可以用于后面的创建文章、获取用户信息等功能。下面是一个 Token 的示例。

```
{"token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c2VyX2lkIjoxLCJleHAiOiJlE0ODEyMTE3NTUsInVzZXJuYW11Ijoicm9vdCIzImVtYWlsIjoiaEBwaG9kYWwuY29tIn0.vd6ubuk9mCL0LTLFs6yGRHJCDCFbFvubCmhn7Ym2YAO"}
```

对一个 Token 进行解析后，就会得到下面的信息：

```
{
  "exp": 1481210927,
  "user_id": 1,
  "email": "h@phodal.com",
  "username": "root"
}
```

在随后的一些需要授权请求里（如创建博客），我们都会带上这个 Token 来进行操作。这个 Token 需要放置在 headers 中，形式是：Authorization: JWT <your_token> 格式。如下是一个创建博客的请求示例。

```
curl -X POST -d '{"title":"test 2","author":"1","body":"test test",
"slug":"post-test"}' -H "Content-Type: application/json" -H "Authorization:
JWT |eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c2VyX2lkIjoxLCJleHAiOiJlE0ODEyMTE3NTUsInVzZXJuYW11Ijoicm9vdCIzImVtYWlsIjoiaEBwaG9kYWwuY29tIn0.vd6ubuk9mCL0LTLFs6yGRHJCDCFbFvubCmhn7Ym2YAO" http://localhost:8000/api/blog/
```

在上面的命令行中传入创建一篇博客所需要的信息和 Token 信息。后台在接受这个请求后，会根据这个 Token 验证是否有效，再根据请求的内容来创建博客。

2. 登录

创建登录页的步骤和之前一样，我们使用生成命令来生成这个页面。

```
ionic g page center
```

并添加 `CenterPage` 到 `app.module.ts` 文件的 `declarations` 和 `entryComponents` 数组中。稍有不同的是，我们想将登录页面放在标签栏上，因此还需要在 `tabs.ts` 中添加一个新的 `tabRoot`，语句如下：

```
...
tab1Root: any = HomePage;
tab2Root: any = BlogListPage;
tab3Root: any = AboutPage;
tab4Root: any = CenterPage;
...
```

并添加相应的入口到 `tabs.html` 文件中：

```
<ion-tab [root]="tab4Root" tabTitle="中心" tabIcon="person"></ion-tab>
```

接下来继续完成登录页面，对 App 来说，也需要相似的 Token 处理过程。

① 登录获取 Token，并保存。

② 在创建博客时，发送 Token。

为了能让用户登录，我们需要创建一个登录表单，代码如下：

```
<ion-content>
  <form (ngSubmit)="loginForm()">
    <ion-item>
      <ion-label>Name</ion-label>
```

```

    <ion-input type="text" [(ngModel)]="user.name" name="name" required>
  </ion-input>
</ion-item>
<ion-item>
  <ion-label>Password</ion-label>
  <ion-input type="password" [(ngModel)]="user.password" name=
"password" required></ion-input>
</ion-item>
<button ion-button type="submit" block>Login</button>
</form>
</ion-content>

```

上面代码中的 `ion-input` 带的 `required` 表示这个元素是必填的。用户未填写就无法提交，并提交相应的信息，如图 9-14 所示。

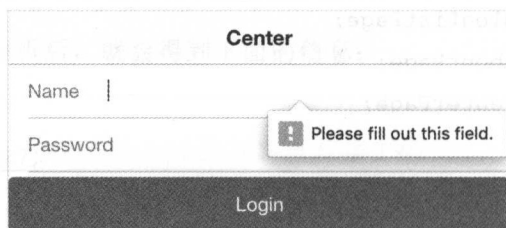


图 9-14 用户名和密码必填的提示

我们会将用户输入的用户名和密码绑定到 `user` 对象上。当用户单击 `Login` 按钮时，就会触发 `loginForm` 函数来处理。这里的 `[(ngModel)]` 则是使用一个双向绑定的语法：当用户在页面上输入值时会改变数据模型，当改变数据模型的时候也会改变页面上的显示。`[(x)]` 语法结合属性绑定的方括号 `[x]` 和事件绑定的圆括号 `(x)`。

然后就可以在 `loginForm` 中处理应用的提交表单的逻辑：

```

export class CenterPage {
  public user = {
    name: '',
    password: ''
  };
}

```

```

constructor(public navCtrl: NavController, private http: Http) {

}

loginForm() {
  let userInfo = {
    username: this.user.name,
    password: this.user.password
  };

  this.http.post("http://localhost:8000/api-token-auth/", userInfo)
    .map(response => response.json())
    .subscribe(
      data => {
        console.log(data);
      });
}
}

```

上面的代码与之前获取博客的代码稍有区别：需要去 POST 数据。因此需要先组装数据，然后通过 http 模板中的 post 方法来发出请求，在请求里需要包含用户名和密码，即上面的 data 对象。如果登录成功，我们会拿到相应的 token，并用 console.log 方法来将其显示在网页的控制台上。

3. 存储登录状态

当我们登录成功后，就会存储 Token 的信息，以便在后面的操作中使用。同时，我们不应该在用户中心页显示相应的登录表单，需要将这个登录表单改成其他内容，如创建博客的表单。

为了使用存储功能，需要使用到 Ionic 的 Storage 组件。Storage 是用于存储键/值（key/value）和 JSON 对象的一种简单方式。Storage 会根据不同的平台来选择对应的存储

引擎。

- 运行原生应用时，Storage 将优先使用 SQLite 来存储数据，它是最稳定，且广泛使用的基于文件存储的数据库。同时可以避免 localStorage 和 IndexedDB 的一些缺陷，如：操作系统将在低磁盘的情况下清除这些数据。
- 当运行在浏览器上或者作为一个 Progressive Web App 应用时，Storage 将按顺序尝试使用 IndexedDB、WebSQL、localStorage。

浏览器上的三种方式比较如下。

- localStorage 适合存储少量的数据。
- IndexedDB 适合大量结构化数据的存储。
- Web SQL 是在 IndexedDB 之前推出的方案，基本上被抛弃了。

在 Ionic 2.0 中，这个存储组件需要另外安装，运行下面的命令来安装这个组件：

```
npm install @ionic/storage --save --save-exact
```

同时，还需要添加到 app.module.ts 文件的 Providers 中，这样才能在模板中使用它，代码如下：

```
import { Storage } from '@ionic/storage';

...

@NgModule({
  ...
  providers: [Storage]
})
```

当用户点击到这个页面时，我们需要去判断用户是否已登录，这时需要查看 Token 是否存储在 LocalStorage 中。因此，我们需要在构造函数中创建一个 storage 对象，并从存储

介质中读取 token 的值。如果存在 token 的值，那么就改变相应的判断状态。

```
public isLogin = false;

constructor(public navCtrl: NavController, private http: Http, public
storage: Storage) {
  this.storage.get('token').then( (token:any) => {
    if(token) {
      this.isLogin = true;
    }
  });
}
```

我们在初始化的时候将 isLogin 的值设置为 false；当我们从存储介质中成功地读取到 token 时，就将 thisLogin 变为 true。与此同时，还需要隐藏登录表单，这时需要使用 hidden 属性来隐藏页面：

```
<form (ngSubmit)="loginForm()" [hidden]="isLogin">
...
</form>
<form [hidden]="!isLogin">
  创建博客
</form>
```

这里的 hidden 属性看上去有点绕，hidden 表示了隐藏条件。当用户登录的时候，!isLogin 的值就为 false，这时创建博客的表单就不隐藏了。

最后，在登录的代码里存储 token 值即可。

```
this.http.post("http://localhost:8000/api-token-auth/", data)
  .map(response => response.json())
  .subscribe(
    data => {
      this.isLogin = true;
```

```
this.storage.set('token', data["token"]);  
});
```

在上面的代码中，我们使用 `Storage` 中的 `set` 方法来存储 `token`。成功存储 `Token` 后，就可以在浏览器的控制台上看到这些数据（打开 `Chrome` 浏览器控制台，点击标签栏中的 `Application`）。当应用在手机上运行的时候，用户是查看不到这些数据的，如图 9-15 所示。

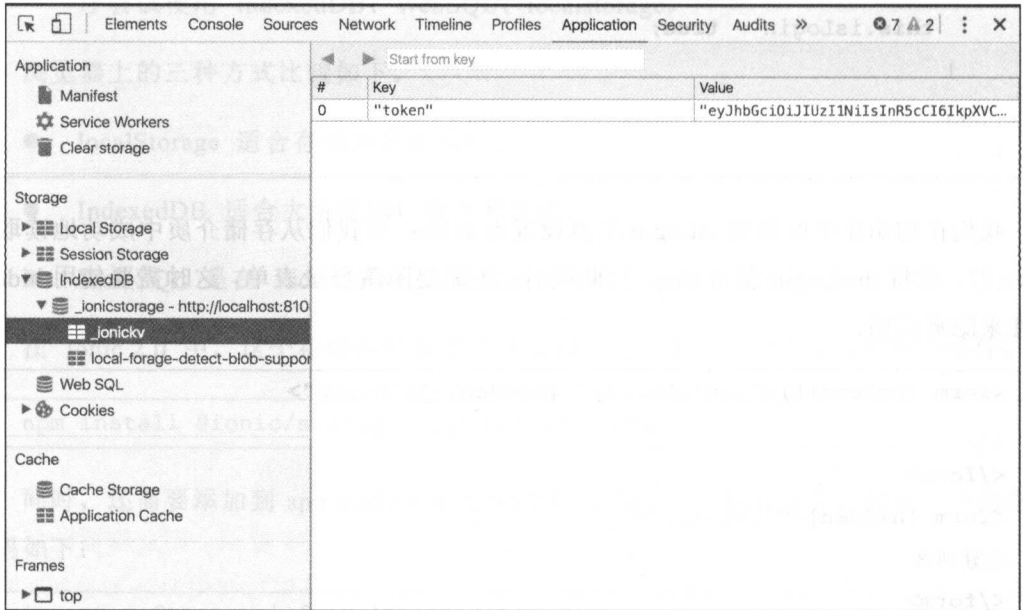


图 9-15 Storage 存储 Token 在 IndexedDB 中

`Ionic` 在 `IndexedDB` 中创建了一个名为 `_ionicstorage` 的数据库，在这个数据库中拥有一个名为 `_ionickv` 的对象存储空间。程序在这个存储空间创建了键值对；获取值的时候，也是从这个存储空间中获取的。

4. 注销当前用户

当应用有登录功能的时候，也一定要需要有注销功能，否则将无法切换为其他用户。实现注销的过程比较简单，只需要以下两个操作即可。

- 将当前的登录状态 `isLogin` 变成 `false`。

- 删除存储中的 `token`。

因此，注销函数就比较简单。

```
logout () {
  this.isLogin = false;
  this.storage.remove('token');
}
```

由于当前的功能比较简单，因此，将“注销”按钮放在页面的 `header` 上。

```
<ion-header>
  <ion-navbar>
    <ion-title>
      Center
    </ion-title>

    <ion-buttons start [hidden]="!isLogin">
      <button end ion-button color="danger" clear (click)="logout()">注销
    </button>
    </ion-buttons>

  </ion-navbar>
</ion-header>
```

相似的，仅当我们登录的时候才会显示这个按钮。

5. 创建博客

有了上面的登录表单的经验后，我们就可以写出下面的创建博客的模板：

```
<form (ngSubmit)="createBlogForm()" [hidden]="!isLogin">
  <ion-item>
    <ion-label>标题</ion-label>
```

```

    <ion-input type="text" [(ngModel)]="blog.title" name="title" required>
  </ion-input>
  </ion-item>
  <ion-item>
    <ion-label>Slug</ion-label>
    <ion-input type="text" [(ngModel)]="blog.slug" name="slug" required>
  </ion-input>
  </ion-item>
  <ion-item>
    <ion-label>内容</ion-label>
    <ion-textarea [(ngModel)]="blog.body" name="body" required> </ion-
textarea>
  </ion-item>
  <button ion-button type="submit" block>创建</button>
</form>

```

在之前讲述 JSON Web Token 的时候，使用命令行来创建博客时，需要添加一个 JWT 的 Headers。Angular 的 HTTP 模板也提供了相应的类，只是过程稍微麻烦一些。

```

let headers = new Headers({ 'Authorization': 'JWT ' + this.token });
let options = new RequestOptions({ headers: headers });

```

最后创建博客的代码如下：

```

createBlogForm() {
  let blogInfo = {
    author: 1,
    title: this.blog.title,
    slug: this.blog.slug,
    body: this.blog.body
  };

  let headers = new Headers({ 'Authorization': 'JWT ' + this.token });
  let options = new RequestOptions({ headers: headers });

```

```

    this.http.post("http://localhost:8000/api/blog/", blogInfo, options)
      .map(response => response.json())
      .subscribe(
        data => {
          console.log(data);
        }
      );
  }
}

```

在上面的代码里，我们将作者的 ID 设成一个固定值，即 1。而这个 ID 值应该是用户的真实 ID，如何获取这个用户 ID？步骤如下。

① 创建一个新的 User API，并发起一个新的 HTTP 请求。

② 解码 Token，从中获取用户信息。

显然可以采用第二种方法：解码 Token。需要寻找第三方插件来解码，angular2-jwt 就是一个不错的选择。安装第三方模块仍然使用的是相似的流程。

安装 angular2-jwt 库：

```
npm install angular2-jwt --save
```

添加到 providers 中：

```

...
import {AUTH_PROVIDERS} from "angular2-jwt";

```

```

NgModule({
  ...
  providers: [
    Storage,
    AUTH_PROVIDERS
  ]
})

```

angular2-jwt 模块提供了一个 JwtHelper 类可以对 Token 进行一些处理：解码 Token、获取 Token 的过期日期、判断 Token 是否过期等。因此，根据它提供的功能进行了一些优化：

①先判断 Token 是否过期，过期时登出。

②从 Token 中取出用户 ID。

因此，最后创建博客部分的代码如下：

```
import {JwtHelper} from "angular2-jwt";

...
public jwtHelper = new JwtHelper();
...

createBlogForm() {
  if(this.jwtHelper.isTokenExpired(this.token)){
    this.isLogin = false;
    return;
  }

  let decodedToken = this.jwtHelper.decodeToken(this.token);

  let blogInfo = {
    author: decodedToken.user_id,
    title: this.blog.title,
    slug: this.blog.slug,
    body: this.blog.body
  };
  ...
}
```

使用 this.jwtHelper.decodeToken 方法来解码 this.token 中的 JSON 数据。数据的格式是

我们之前在讲述 JSON Web Token 时看到的：

```
{  
  "exp": 1481210927,  
  "user_id": 1,  
  "email": "h@phodal.com",  
  "username": "root"  
}
```

其中的 `user_id` 就是当前登录用户的 ID，用这个 ID 替换之前固定的值 1，就可以完成功能。也可以通过这个插件来优化跳转到页面时的权限认证。有兴趣的读者可以参考本书配套的代码，这里就不展开介绍。

9.4.2 测试和发布应用

当我们开发完应用后，就可以在移动设备上进行测试，并着手准备发布的过程。首先需要在相应的操作系统上安装相应的 SDK。

- 测试、发布 iOS 时，需要在 Mac OS 上安装 XCode，并购买相应的开发者证书。
- 测试、发布 Windows Phone 时，需要在 Windows 10 上安装 Visual Studio，并购买相应的开发者证书。
- 测试、发布 Android 时，可以在不同的平台上（Windows 10、Mac OS、Linux）安装 SDK，并申请对应的应用商店的账号，如 Google Play、豌豆荚、应用宝等。

因此，如果计划发布三个平台的应用，就需要至少一个 Mac OS 系统的机器和一个 Windows 系统的机器。

为了构建不同的平台应用，需要添加不同的平台，如：

```
ionic platform add android
```

上面的命令可以为项目添加 Android 平台的支持，过程如下面的日志所示：

```
Adding android project...
Creating Cordova project for the Android platform:
  Path: platforms/android
  Package: io.ionic.starter
  Name: V2_Test
  Activity: MainActivity
  Android target: android-23
Android project created with cordova-android@5.1.1
Running command: /Users/fdhuang/repractise/growth-blog-app/hooks/after_
prepare/010_add_platform_class.js/Users/fdhuang/repractise/growth-blog-app
```

最后，再执行 `run` 就可以在对应的平台上运行，如：

```
ionic run android
```

构建时，只需要运行 `build` 命令即可：

```
ionic build android
```

我们还需要准备图标和启动页的图片，而 `ionic` 命令提供了一个 `resources` 工具可以帮助我们解决不同尺寸的图片问题。只需要保存 `icon.png`、`icon.psd` 及 `icon.ai` 到 `resources` 目录下，最小的大小是 `192 px × 192 px`，并且没有圆角。

```
ionic resources --icon
```

同理，启动页图片，保存 `splash.png`、`splash.psd` 或者 `splash.ai` 文件到 `resources` 目录下：

```
ionic resources --splash
```

`Ionic` 将会自动创建不同分辨率和不同大小的资源文件，并自动修改 `config.xml` 文件。

最后，再根据不同平台的发布流程进行发布即可。

9.5 小结

本章介绍了一系列关于前端及混合应用开发相关的知识。从单页面应用的基本技术开始，介绍了诸如 DOM 与事件、DOM 与事件、控制器与状态、路由与页面等知识。这些知识构建了前端应用的基本骨架，理解它们有助于更好地开发应用。随后对于前后端交互所需要的 API 知识进行了详细介绍。同时，还对主流的单页面应用进行介绍。

接着，开始本章最主要的内容：创建移动应用。先介绍了混合应用框架 Ionic、如何使用它来创建应用，以及它的初始化过程。随后使用 Ionic 来对接之前实现的博客应用，并介绍了如何创建相应的 API。然后介绍了如何使用 JSON Web Token 进行授权。最后，介绍了如何用它来实现用户登录，以及借助于 Token 来创建博客。

由于本章涉及的知识比较多，建议读者可以阅读相关的资料，并多加练习。

推荐阅读

- Angular 2 中文文档。
- Ionic 2 官方文档。

第9章花了相当长的时间介绍了编写混合应用，然而这个系统很明显已经成了一个遗留系统（Legacy System）。我们刚写新的代码怎么能称为“遗留系统”呢？

在我刚开始工作的时候，我以为编程就是不断地添加新的功能，当时从未想过需要处理大量的遗留代码。然而，在经历了一系列对旧代码、遗留代码维护的时间后，我发现处理旧的代码更具有挑战性。你可以很容易地在你的书架上找到一本如何编写某个框架、语言的书，但是你却很难找到一本教你如何修改现有代码的书。

你可以想象这样一种场景：当你更换到其他项目时，发现他们的代码没有注释、文档，而且代码质量很差。你可以想象这样一种场景：当你接手一个遗留系统时，总在担心你修改的部分可能会影响其他代码，特别是那些没有测试的代码里。

第3部分 增量性优化

同样，这种系统也存在于各种各样的公司里：互联网公司、大公司、初创公司。软件开发人员在软件开发过程中总会不断地重构。在这个流动过程中，增加新知识、业务逻辑之消耗，代码就会因此变得越来越难以改动，这时一些好的方式就是重构。重构，即在不改变程序逻辑的前提下，对程序代码进行改造，它需要花费更多的成本。除了重构，你还可以选择其他的技术方案。

本章将介绍：

- 遗留代码：如何识别、如何重构、如何测试、如何部署。
- 如何提高代码质量：如何识别、如何重构、如何测试、如何部署。
- 如何使用工具对代码进行重构。

这些内容会在我们的编程生涯中不断出现，有时甚至会让你无能为力——重构还是重构是一个很难的问题。因此，现在让我们逐一了解这些知识。

第 10 章

遗留代码与重构

本章将引入遗留代码的概念，并介绍遗留代码及遗留系统是如何一步步形成的，以及我们在工作过程中是如何引入技术债的概念来保持对系统的警惕的。最后，我们将介绍重构，并使用重构来改善软件的质量。

第 9 章花了相当长的时间介绍了编写混合应用，然而这个系统很明显已经成了一个**遗留系统**（Legacy System）。我们刚写完的代码怎么能称为“遗留系统”呢？

在我刚开始工作的时候，我以为编程就是不断地添加新的功能；当时从未想过需要处理大量的遗留代码。然而在经历了一段对旧代码、遗留代码维护的时间后，我发现处理旧的代码更具有挑战性。你可以很容易地在你的书架上找到一本如何编写某个框架、语言的书，但是你很难找到一本教你如何修改现有代码的书。

你可以假想这样一种场景：当你更换到其他项目组时，发现他们的代码没有注释、文档和测试用例，也没有人可以告诉你可以怎么启动应用等。你需要自己去阅读代码，然后一步步地了解业务与技术等。这时可以将这样的系统称之为遗留系统。当你在修改这样的系统时，总在担心你修改的部分可能会影响其他代码，特别是那些没有测试的代码里。

同样，这种系统也存在于各种各样的公司里：互联网公司、大公司、硬件公司等。软件开发人员在软件开发过程中总会不断地流动，在这个流动过程中，很多知识、业务也随之消散。代码就会因此变得越来越难以改动，这时一种好的方式就是**重写**。还有一种方式也是**重构**，它需要花费更多的成本。除了重写与重构，我们在平时的软件开发过程中应该持续关注项目的技术债务。

本章将介绍：

- 遗留代码、遗留系统，以及它们是如何形成的。
- 如何提高代码质量。
- 如何使用 IDE 对代码进行重构。

这些内容会在我们的编程生涯中不断出现，有时甚至会无能为力——重写还是重构是一个很难的问题。因此，现在让我们逐一了解这些知识。

10.1 遗留系统

相信你也经常看到关于“某某网站的架构之路”的文章，会发现其中一个很有趣的过程，就是：它们会把之前的架构抛弃掉；接着又做了一个这样的系统，多年后这个系统又被重做了。只要这个应用还存在，它就会不断地重复这个过程。因为在这种情况下最好的选择是让旧的系统继续运行，而新的系统也在不断开发着。既然它在旧的系统中工作得很好，那么我们就没有理由去修改它们。当有新的需求出现时，可以重新设计一个新的系统。

但是为什么这种情况（重写系统）在不断地重复着呢？

10.1.1 什么是遗留系统

维基百科上对遗留系统的定义是：

一种旧的方法、旧的技术、旧的计算机系统或应用程序。

实际上，遗留系统并不仅仅局限于使用旧的技术、方法，还有诸如没有人知道业务的历史遗留代码。当你看到某个网站宣称用新的框架来替换旧框架时，你应该知晓它们原有的系统是遗留系统。人们已经不想在上面工作了，很多代码也不知道是干什么的，也没有人想去深究——毕竟不是自己的代码。

那些让人无从下手的系统、代码里包含了下面一些原因。

- 几乎无法维护，难以修改。尽管我们仍然有能力对我们的代码进行修改，但是我们会遇到一系列的问题，如对业务不了解，改动引起的变化太大。
- 代码遗失、子模块无法使用。对一些依赖于诸多第三方模块的、中大型的系统来说，这种问题很容易遇到：当你试图去构建一个旧的系统时，发现它依赖于一个第三方模块，这个模块又依赖于一个额外的第三方模块，然而它已经过时了或者没有办法使用。

- 没有文档或者过于简单。当我们手上没有一份文档时，就需要深入代码去理解系统。而如果代码本身又不容易读懂，就需要花费巨大的时间与精力才能理解系统。
- 代码逻辑不清晰。这种情况特别容易发生在阅读一个编程经验比较少的程序员写的代码里。当有了一定的编程经验之后，我们的思维方式和之前就大有不同。

除此之外，我们还可能遇到其他一些状况，如无用的代码等。我们都知道“千里之堤，溃于蚁穴”，遗留系统的形成都是遗留代码造成的，一行行难以维护的代码造成系统难以维护。

1. 遗留代码

对于遗留代码，不同的人有不同的看法，如：与我们生活息息相关的软件中，很多满是错误、脆弱的代码，并且这些软件难以扩展。在一些极端的例子里，人们没有自动化测试的代码就是遗留代码。

从一个新手程序员到一个老鸟，我们的编程水平都在不断增加。在过去写的代码一直都在那里，我们一直都没有足够的勇气去修改它们。因为我们知道如果一不小心改错了，就会导致一些意外的 Bug。这些 Bug 可能会对我们的编程生涯造成一些影响。我们不知道这样做的后果，是因为我们没有对原来的代码进行测试。如果代码都是经过测试的，那么在修改中出的错都会在测试中加以体现。

虽然在这些代码里可能拥有完整文档、依赖管理很好、逻辑清晰，然而它没有测试。因为没有足够的测试覆盖，没有人敢去修改这些代码，这些代码就很容易变成遗留代码。

在第5章中，我们提到不写测试的原因，除了原有的软件工程软件不佳，还有可能是上线时间导致的。而对于遗留代码，我们也很容易找到一些相似的原因，这些原因很容易与没有编写测试有关系。

(1) 交付压力

对一个明天就要上线的代码来说，你本身可能就没有足够的时间来完成功能，更何况是编写测试呢？这时不得不对此做出妥协。而如果这时再遇上紧急上线，那么就不得不继续赶工。在这时不要说是测试，就连代码质量本身也是一个问题。任务越是紧急，代码就

越容易出现 Bug，就越容易陷入恶性的循环：没有测试、充满 Bug、赶工的同时修 Bug。与此同时，我们也没有足够的时间去学习来提高技能。

（2）技能不足

回过头去看看我们在编程初期写的那些代码里也充满了各种“坏味道”（Code Smell），如各种神奇数字、奇怪的变量名、函数名等。如果我们所在的项目组没有代码审查，那么写的这些代码就会进入代码库里。我们所说的这些内容都是一些基础部分，我们还可能设计出一些奇怪的 Bug、没有缓存带来一些性能问题等。这些都是因为技能不足，才写出这些难以维护的代码。

当我们不断地进行反馈性的编程练习时，我们就可以通过此来做出改进。在我的职业生涯初期，最常遇到的问题是：因为编写测试能力的不足，有时会跳过一些代码的测试。

（3）功能复杂

代码复杂的原因是：业务功能复杂。尽管复杂的业务功能本身是存在一些问题的，如时间一长，业务人员也会忘记原本的业务功能。但是我们并不会在这里讨论业务的原因，毕竟复杂的代码总会存在的。复杂的代码可能意味着：多层嵌套、长的函数等代码问题。除此之外，它也意味着我们在编写测试上会遇到一些麻烦。

要对付功能复杂的代码最好的方式是：针对各种情况编写测试，其次才是编写相关的文档。测试才是功能正常的最好保证，当我们修改了一种情况时，只应该是这种情况的测试失败。文档无法直接告诉我们修改到的这个代码可能会影响到其他功能。

因此，判断是否是遗留代码的条件很简单：**维护成本是否比开发成本高很多。**

2. 造成遗留系统的因素

当一小部分难以维护的代码开始扩散时，系统就开始变得难以维护。如果代码没有出现上面的问题，我们的系统可能表现得还不错。

幸运的是，多数系统并不存在遗留代码导致系统遗留的问题——因为多数应用的生命周期并没有那么长。特别是在最近几年的技术发展里，每年都要应对新的设备、设计新的

接口，这时影响我们系统的主要因素变成了技术栈和可扩展性。

可扩展性在技术上的体现就是软件的架构：如何支持更多的访问用户。要让更多的用户访问我们的网站，即我们需要提高网站的并发数。提高并发量有两种相去甚远的做法横向扩展和纵向扩展。

（1）纵向扩展

纵向扩展就是使用更好的机器来提高并发数。纵向扩展是一种较简单的扩展方式，我们只需要购买一个更好的机器就可以。如果我们可以估计网站用户人群的数量，并且这个数量级远远小于好的服务器所能承受的，那么这就是一个可以快速采用的方式。由于它可以在短期内实现更多的并发数，并且它不需要花费额外的开发时间，因此很容易看到它被采用。

并且我们知道单个服务器存在极限，如内存、CPU、硬盘上等，这时就需要考虑使用横向扩展。在我经历的一个项目里就遇到内存上的极限值问题——由于操作系统限制，已经无法使用更大的内存，而我们的应用仍然需要更高的内存来处理。

（2）横向扩展

横向扩展就是使用更多的机器来提高并发数。尽管使用横向扩展是一种正确的方式，但它要实现起来并不是那么容易。当我们开始在多个机器上运行应用的时候，来自多个不同服务器的请求就会抢占资源，如数据库。这时就需要引入“锁”的机制，当有一个任务占用资源，就需要锁住资源。而随着引入“锁”的机制，系统就会变得愈加复杂。因此，当我们讨论应用的架构设计时，很容易就会进入相关内容的讨论。

对大部分的应用来说，已经有足够多的方案可以让你支持更多的用户。这些知识很容易就能从不同的文章、书籍中获取到。然而，可扩展性不仅仅体现在技术上，还体现在组织结构设计上。在我们的应用需要支持更多的用户时，也意味着我们的团队需要更多的开发人手。这时就涉及如何让团队变得可扩展，就需要开始关注下面一些内容：

- 人员流动。这是一个不可避免的问题，如果一个团队成员的变化太大，那么它在运行上就会存在一些风险。当有过半的成员离开项目时，这就意味着我们可能没

有足够的力量来应用这种变化。理想的团队应该可以实现：添加一倍的成员，即一个有经验的人可以带一个新人成长。如果在短时间内添加过多的人手，那么就没有足够多的人手来带领新人成长。

- 知识分享。我们在团队里需要有一些好的机制来进行业务、技术相关知识的分享，它可以帮助团队里的新人更好地成长，又可以让旧的知识得以传承。它可以让团队里的知识尽可能多地共享，并减少因有经验的开发者离开带来的问题。

在团队里内部应该鼓励技术分享、业务知识分享等，它既可以提供项目成员的技术水平，又可以让项目相关的领域知识传递下去。它可以避免当有项目成员离开时，大量的领域相关的知识也随着离开。

除可扩展性外，造成遗留系统还有一个主要原因是：技术栈。技术栈影响系统则存在着多种原因，常见原因如下：

- 原有的技术栈已经被遗弃。我们过去所采用的技术栈在今天已经不维护了。多数情况下，我们只能使用新的技术栈开发一个新的。运气好的话，我们只需要更换相应部分的组件即可。
- 原有技术栈的开发成本太高。这里的开发成本指的不一定是技术栈本身，还有可能是相对于新的技术栈来说，又或者是很难招到相应技术栈的人才。
- 原有技术栈不能满足新的需求。
- 战略性迁移技术栈。尽管系统本身不是遗留系统，但是它会在内部被视为“遗留系统”。

这些技术栈相关的因素都会在不同的程度上影响着系统的未来，也是我们不得不经常面对的一些问题。在了解了遗留系统的成因之后，让我们看看如何采用一些策略来减缓。

10.1.2 遗留系统改造

要对遗留系统进行改造不是一件容易的事，因为我们要面对缺乏领域相关知识和技术

栈相关知识等问题，我们还会面临着一些我们不了解的技术细节带来的问题。在这里，我们就不讨论从业务人员、开发人员、文档中获取现有业务知识的相关内容。我们只讨论如何去阅读遗留代码，并尝试去改造现有的遗留系统。由于笔者经验有限，这里仅介绍一些我做过、尝试过的方法。

1. 遗留系统改造

按照上面的观点，在修改代码之前，我们应该先为应用添加测试。随后对于应用的改造流程，应该按本书在之前提出的步骤一步步向下执行：

- 添加自动化构建。
- 添加测试框架。
- 添加一些重要的测试。
- 使用持续集成。
- 添加测试，并提高测试覆盖率。

当我们想要修改的代码都有测试覆盖的时候，我们就可以考虑对代码进行重构。我们将在本章的剩余部分介绍一些基本的重构技巧。

如果系统相当复杂，我们就可以考虑将系统拆解为多个服务，然后舍弃现有的系统。

2. 更新技术栈

如果是针对技术栈来进行升级，那么它看上去就相对会比较容易——如果我们已经做好了上面的相关实践。不过，首先需要保证你的代码具有版本管理的功能。如果没有，则立即创建一个，要优先保证本地的修改都是可以恢复的。如果我们只是想更新一下所使用的技术栈的版本，如更新 Django REST Framework 的版本，那么应该这么做：

- 寻找使用到的代码。
- 检查对应版本的 Release Note。
- 按需要更新代码。

- 无法更新则回滚。

如果要使用一个新的技术栈来替换旧的技术栈，就不会那么轻松了——我们需要重写这部分内容。这也就是为什么我们在技术选型与业务中花了很多篇幅来介绍技术选型的相关事项。这种经历相当痛苦，甚至你还会花费大量的时间来犹豫是否更新，又或者是否来重写这部分业务。

3. 技术债

在平时的工作中，我们会阅读一些相关的技术周报、技术文章、技术新闻等内容。在这些技术资料中，我们很容易关注到所使用的库、依赖、模块的版本，然而限于交付压力的原因，我们并不会立即升级这个软件的版本。这时我们就会将其称为**技术债务**，简称技术债。在更广泛的领域里，技术债就是一些实现不好，但是可以在更短的时间内实现功能的方案，而且它有一种更好的方案，只是需要花费更长的时间。

我们所使用的技术栈比较老旧，也可以将之归到技术债里。除此之外，还有诸如编写的代码没有测试、测试覆盖率不够等。只是由于我们的交付时间有限，我们并不能将这些东西做好。

因此，建议读者在平时遇到类似的技术债问题时记录下来。在必要的时候，我们应该偿还这些技术栈，它能避免让我们的应用变成遗留系统。

在平时的工作中，技术债就是写得不够整洁的代码，它们需要重构来改善质量。

10.2 易读的代码与重构

过去，我有过在不同的场合吐槽别人的代码写得烂，而我写的仅仅是比别人好一点而已——而不是好很多。要讨论如何写好代码是一件很难的事，人们对同一件事物的考虑都是不一样的。同样的代码在相同的情景下，不同的人会有不同的设计模式。我们没有办法在这里讨论设计模式，也不需要讨论。

我们所需做的最简单的事是确保我们的代码易读、易测试，因为代码是写给人看的。因此，这里只介绍四个基本的要素：

- 确保我们的变量名、函数名是易读的。
- 一个函数只做一件事。
- 减少重复的代码。
- 易于阅读的排版。

然后再去测试它，这样你就知道需要的是什么功能。如果我们没有足够的时间去测试，就应该尽可能地保证代码是易读的。

10.2.1 命名

命名是一个特别长的，也是特别忧伤的故事。我想作为一个程序员的你也相当恐惧这件事。一个好的函数名、变量名应该包含这个函数的信息，如这个函数是干什么的，或者这个函数是怎么来的，这个变量名存储的是什么。

正因为取名字是一件很重要的事，所以它也是一件很难的事。一个好的函数名、变量名应该能正确地表达出它的含义。如你可以猜到下列代码中的 `i` 是什么意思吗？

```
fruits = ['banana', 'apple', 'mango']
for i in fruits:
    print 'Current fruit :', i
```

如果换成下面的代码会不会更容易阅读呢？

```
fruits = ['banana', 'apple', 'mango']
for fruit in fruits:
    print 'Current fruit :', fruit
```

命名不仅存在于变量名上，还存在于对函数的命名上。如我们可能会用 `getNumber` 来

表示获取一个数值，但是要知道这样的命名并不是在所有的语言中都可以这样用。如在 Java 中存在 `getter` 和 `setter` 这种模式，代码如下：

```
public String getNumber() {  
    return number;  
}  
  
public void setNumber(String number) {  
    this.number = number;  
}  

```

如果我们是取某个东西的数值，应该使用 `retrieveNumber` 这样更具代表性的名字。

在《编写可读代码的艺术》中，介绍到了关于命名的几个注意事项：

- 选择专业的词。最好是可以和业务相关的，它应该极具表现力。
- 避免像 `tmp` 和 `retval` 这样空泛的名字。不得不提到的一点是，`tmp` 实在是一个够烂的名字，将其变为 `timeTemp` 或者类似的会更直观，`tmp` 只应该是名字中的一部分。
- 用具体的名字代替抽象的名字，它可以更好地描述事物。
- 为名字赋予更多的信息。
- 变量名不宜过短，也不宜过长。这就需要依照自己的经验来好好把握了，太长容易因为换行导致不易阅读，太短则不能表达与代码相关的含义。
- 利用名字的格式来传递含义。如全大写来表示常量，类的首字母应该是大写，而变量的首字母则不能是大写。

更多详细的内容，建议读者阅读相关的书籍。

10.2.2 一次只做一件事

本书一直强调 Tasking（任务拆分）的原因就是：一次只做一件事。同样，对于代码来说，我们也应该这样做。

函数是指一段在一起的、可以做某一件事的程序。

从定义上说，一段函数应该只做一件事——但是什么才是真正的一件事呢？实际上还是 Tasking，将一个复杂的过程一步步地分解成一个个函数，每个函数只做它的名称对应的事。对于一个具体的功能来说，它有一个固定的过程，在这个过程中的每一步都是一个函数。这些做某件事的代码变成了一个函数，这个函数负责相应的业务逻辑。

当某个函数的业务逻辑过多时，这个函数就可能相当长。长的函数会花费大量的代码阅读时间，并且要测试长的函数也不是一件容易的事。而如果我们代码没有进行测试，这些代码就会变得越来越难以理解。在上面的章节里，我们说过**没有测试的代码也将会变成遗留代码**。除此之外，在代码较长的函数中，我们很容易就会发现它们有嵌套过深的问题。

函数的功能一般都会体现在函数名上。因此，从函数名上就可以看出一个函数是否做了很多事。如下的函数名就很明显：

```
function checkAndSubmit(){  
    ...  
}
```

我们可以将其拆成三个方法，每个方法都足够小，并且只做一件事。

```
function checkAndSubmit(){  
    check();  
    submit();  
}  
  
function check(){  
    ...  
}
```

```
function submit(){
    ...
}
```

在实际工作中会遇到比这个例子更复杂的情况，要判断一个函数是不是做了很多事情，除了上面说的函数名，还有一个比较直观的做法是：看函数的长度。大多数情况下，长的函数意味着我们在这个函数里做了很多事情。当然，如果这个函数中有很多变量声明，那么就在这个没有关系了。

10.2.3 减少重复代码

代码重复的一个主因是：复制/粘贴——毕竟这是实现某个功能最快的方式。它的主要问题是，在需求发生变更的时候会发生问题。当某个需求变更，我们就需要去一处处地修改代码；如果需要修改的地方比较多，则有可能因此而漏掉一些需要修改的部分。除此之外，还会影响我们阅读代码时的体验，我们总会不自觉地去比较代码中的不同之处，了解为什么这里会出现这样的代码——是不是手误了？

下面的代码用于切换标签栏的函数：

```
function setTab(index) {
    this.currentTab = index;

    this.currentCategory = CATEGORY[this.currentTab];
    Category.get({category: this.currentCategory}).$promise.then(function
(data) {
        // ...
    })
}

function nextTab() {
    this.currentTab = this.currentTab + 1;
```

```

    this.currentCategory = CATEGORY[this.currentTab];
    Category.get({category: this.currentCategory}).$promise.then(function
(data) {
    // ...
    })
}

function prevTab() {
    this.currentTab = this.currentTab - 1;

    this.currentCategory = CATEGORY[this.currentTab];
    Category.get({category: this.currentCategory}).$promise.then(function
(data) {
    // ...
    })
}

```

上面的三个函数一共有三种不同的行为：点击当前标签的 `setTab` 函数、切换到下一个标签的 `nextTab` 函数、切换到上一个标签的 `prevTab` 函数。从代码中可以发现，这个函数中有相当一部分的语句是重复的。由于在可预见的未来不会有太多的变化，因此，我们可以提取公共部分为一个函数。这样做既可以减少代码重复，又可以方便我们一次修改多处代码。

然而，DRY（Don't Repeat Yourself）原则是特别值得玩味的。当我们不断偏执地去减少重复代码的时候，会导致代码的复杂度提升。当业务发生一些变更时，我们还需要去拆解这些重复的代码——将其拆解为两个函数。为了不重复代码，而提取不同功能的代码是一种不理智的行为。

在多数时候，我也喜欢用复制/粘贴功能，然后再针对性地修改内容。界定我们是否需要提取相同部分的原则是：事不过三。当一段代码的重复频率达到了三次，我们就需要开始考虑对它进行重构。

10.2.3 排版

在看到代码的一瞬间，我们就会决定要不要去读它。如下是一个名为“feature.js”的库中的部分代码：

```
!function(e,t,n){"use strict";var r=t.documentElement,i={  
  create:function(e){return t.createElement(e)}...
```

上面的代码是经过压缩后的部分代码，当你看到这样的代码时，我想你不会有继续阅读下去的兴趣了。这个例子是一个极端的例子，但是在我们的周围中有很多类似的情况，如：混合了匈牙利命令法、驼峰命令法、下画线等几种不同的命名法则，使用了不同的缩进格式等。

在 C 语言里，你可以自由地在“;”号后加上其他语句，或者换句再添加别的语句。而在 Python 语言里，你就不能这样做了——虽然没有“;”号，也需要严格地按照格式来。这也是为什么 Python 语言看上去比较简单的原因：你看不到混乱的代码排版。

人们对代码在屏幕上的呈现形式有不同的喜好，这一点不仅体现在使用编辑器 / IDE 的主题色彩，还有各种不同排版的争议——如在 Python 语言里使用 Tab 键，还是使用 4 个空格来表示缩进；在 JavaScript 中使用 4 个空格还是使用 2 个空格来表示缩进。

讲究排版的目的是让我们阅读起来更轻松，更符合习惯。我们会在函数的代码间添加空行来区分不同的代码块——如函数声明和方法操作，如在之前编写的登录函数：

```
let headers = new Headers({ 'Authorization': 'JWT ' + this.token });  
let options = new RequestOptions({ headers: headers });  
  
this.http.post("http://localhost:8000/api/blog/", blogInfo, options)  
  .map(response => response.json())  
  .subscribe(  
    data => {  
      console.log(data);  
    });
```

代码中的变量和登录部分以空行区分开，我们还可以修改代码中的 `blogInfo` 变量：

```
let blogInfo = {  
  author : decodedToken.user_id,  
  title  : this.blog.title,  
  slug   : this.blog.slug,  
  body   : this.blog.body  
};
```

让代码中的值可以对齐，以便能一眼看出这些值是否正确。

在个人的项目中使用哪种换行模式并不重要，使用哪种空格来表示缩进也不重要，重要的是你要保持风格的一致性。而如果这是一个团队项目，就需要讨论出一套合适的代码风格，并强制性地让成员都遵守这些风格。一种比较简单的做法是直接寻找成熟的风格方案，如 PEP8、《Google 开源项目风格指南》等。

10.2.4 重构

要解决遗留系统的问题，除了编写出容易阅读的代码，我们还应该学会对原先的代码进行重构。

在上面的部分里，我们介绍了一系列关于如何编写阅读代码的技巧，这些知识也是重构代码的基础。关于重构的更多知识，建议读者阅读《重构：改善既有代码的设计》。这里只是简单介绍如何使用编辑器或者 IDE 中的工具进行重构。

大部分开发工具都有相应的重构插件可以使用，如在 Visual Studio Code 中可以使用：JS Refactorings 插件，它可以提供诸如：

- Convert To Member Function
- Convert To Named Function
- Export Function

- Extract Variable
- Shift Parameters Left
- Wrap in function
- Wrap in IIFE

等重构功能，我们只需要在重构的代码上按下快捷键，就可以对代码进行有选择地操作。笔者习惯使用 JetBrains 系列的 IDE 来进行重构——由于之前使用 IntelliJ Idea 编写的 Java 代码，并接受了相应的培训。在这个系列的 IDE 里，与重构相关的内容都放置在如图 10-1 所示的独立菜单中。

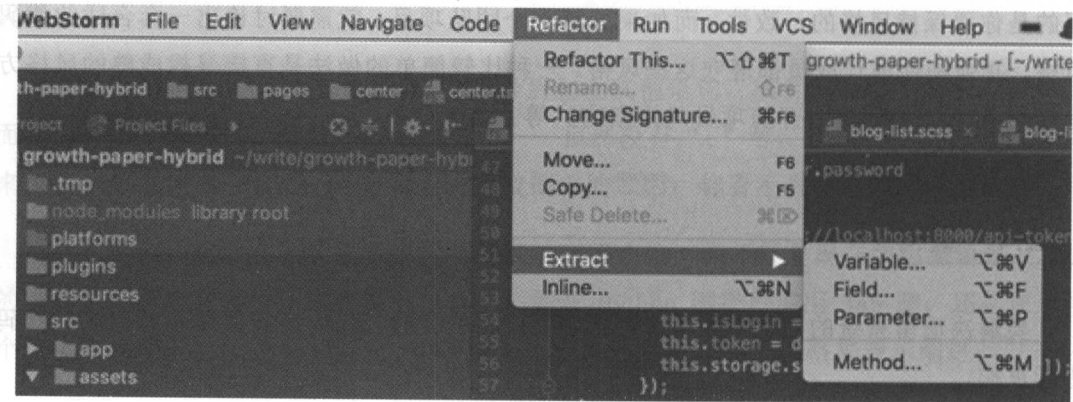


图 10-1 WebStorm 重构菜单

不过在 WebStorm 中，我们能使用的重构功能是有限的，在 IntelliJ Idea 中可以使用差不多 30 个类似的重构选项。

在这些重构技能里有以下几个比较常用。

1. 重命名

当我们刚写下一个函数或者变量的时候，可能会使用一个临时的名称来表示含义，随后会重新修改这个变量的值。如果没有重构相关的功能，我们就需要一个个手动复制、替换，或者全局替换这些变量。而如果我们开发工具支持这样的操作，那么我们就可以直

接按下 `Shift + F6` 组合键对其进行重命名。

2. 提取函数

当我们在一个函数里写了相当长的代码时，我们就会考虑提取部分代码成一个新的函数。这时遇到的一个问题是要提取的代码中包含了一系列的变量，这些变量在手动编写时又容易出错，因此，我们同样可以使用 IDE 来进行重构。

- 选中需要提取的部分。
- 单击 Refactor 菜单中的 Extract 下拉选项，再选择其中的 Method。
- 在弹出的弹框中输入相应的变量名和函数名即可。

这个重构方式可以用于改善在上一节中提到的“长的类”。

3. 提取变量

在《重构》一书中的叫法是：引入解释性变量，解释如下：

将该复杂的表达式（或其中一部分）的结果放进一个临时变量，以此变量名称来解释表达式用途。

同样，要使用 IDE 来进行这个重构也相当容易，可以使用菜单进行重构，也可以使用快捷键来重构（显示在菜单上）。提取变量特别适合于：提取条件语句中的条件，将其提取成一个有含义的变量。如下是之前用于判断 Token 是否过期的代码：

```
if(this.jwtHelper.isTokenExpired(this.token)){
    this.isLogin = false;
    return;
}
```

我们可以直接提取条件：

```
let isTokenExpired = this.jwtHelper.isTokenExpired(this.token);
if(isTokenExpired){
    this.isLogin = false;
```

```
    return;  
}
```

当其他人看到这部分代码的时候，他们就不需要去阅读相应的条件代码，直接看 `isTokenExpired` 就可以知道需要的条件是什么。

这里只是做一个简单的入门介绍，更多的内容还需要交由读者去探索。

10.3 小结

本章介绍了遗留系统中形成的一些因素，以及其中一些不可避免的原因。我们还简单介绍了如何对遗留系统进行改造，以及对遗留的技术栈进行升级。

当我们意识到：一个系统的代码质量需要去提高的时候，我们需要拥有好的代码意识，还需要有基本的技巧来帮助我们重构代码。因此，我们还向读者展示了容易阅读的代码的一些基本特性，以及使用 IDE 来重构代码。

在编写更好的代码时，我们可能就会想到设计模式。设计模式是对软件设计中普遍存在（反复出现）的各种问题所提出的解决方案。通过以往的经验，我们就能依据一个环境来识别一个模式。不过，遗憾的是设计模式依赖于整个团队的水平。对于了解设计模式的人来说，设计模式就是一种沟通语言；而对于不了解设计模式的人来说，设计模式就是复杂的代码。

模式和重构之间存在着天然联系，模式是你想到达的目的地，而重构则是从其他地方到达这个目的地的条条道理——Martin Fowler《重构》。

我们没有对设计模式介绍的一个原因是——它需要有大量的编程经验，才可以让我们实现：重构到设计模式。

要么我们在遗留代码扩散趋势开始之时，有针对性地对其制止；要么我们就做好准备设计新的系统。在下一章里，我们将介绍关于架构重新设计，并回顾一些基础知识。

参考阅读

- 《重构：改善既有代码的设计》。
- 《修改代码的艺术》。
- 《软件设计重构》。
- 《编写可读代码的艺术》。

第 11 章

增长与新架构

本章将介绍敏捷回顾的概念，并介绍这样的反馈系统将如何改善团队交付的软件质量。同时，我们还将回顾前面的章节，对其中做得好的和不好的地方进行总结。然后介绍如何去构建自己的知识体系。最后将介绍如何去设计新系统的架构。

每年年底的时候，我们都会看到形形色色的总结，有的是公司的，有的是个人的，有的是关于工作的，有的是关于生活的等。在这些总结里，我们都会看到他们在点评这一年的得与失——哪里做得好，哪里做得不好。根据过去一年的得失来思考未来一年，并做一些改进。对项目来说也需要采用类似的机制，只是这个周期比较短，即我们之前说的迭代。因为对大部分项目来说，一年的周期实在太长，很难依据反馈改善当前的产品。

在敏捷团队里，每个迭代结束的时候，都会有一个回顾会议。在这个回顾会议里，我们会总结这个迭代做得好的地方和一些实践不好的地方。对于好的实践则会进行下去，而不好的内容则会进行改进。在很多项目里，我们同样也会看到最后都有类似的总结。也只有总结了，我们才会成长。

因此，本章将介绍：

- 敏捷回顾，以及它是如何帮助团队成长的。
- 结合搜索引擎与书籍构建知识体系。
- 如何设计一个新的架构。

技术本身是在不断推陈出新的，而这个过程只有学习技能是不变的。只有在我们的祖父辈时代才能仅凭一门技能、一个技术栈就能生存下去。在技术领域里，技术的时效性变得越来越短——你可能在年初使用了这个框架，在年底的时候，这个框架已经差不多被社区抛弃了。而好在新的框架都基于旧的框架，或是在旧的基本上添加新的特性，或是结合几种不同的理论而成。

在平时学习时，我们使用书籍来快速入门；在工作使用时，我们需要结合搜索引擎；在业余深入时，我们则进入代码的底层探索。在这个过程中，我们所做的事件就是在不断地学习，并生成知识体系的索引。

对我们而言，我们需要回顾旧有的知识，才有能力去改变。在开发软件时，我们使用重构和技术债来改进软件；在团队协作时，我们使用敏捷回顾来帮助团队更好地成长。

11.1 增长

在讨论迭代和精益的时候，我们实际上是在讨论反馈。在之前的章节里，对读者来说就是一个输入知识的过程，而对笔者来说则是一个输出知识的过程。然而，这只是表面上的一个过程，实际上，对笔者来说，在输出的过程中也在不断地输入知识。

当我们学习一个新的技术栈时，便是在输入某个技术栈的知识。当我们出于工作的需要来学习某个技术栈时，对这个技术栈的了解就是有限的。当我们以写作等输出知识的方式总结自己学到的东西时，就会比我们使用这个技术栈深入学习更多的内容。这时学习到相关的知识点内容就更加深入，因此，我们常说：**输出是最好的输入**，如图 11.1 所示。

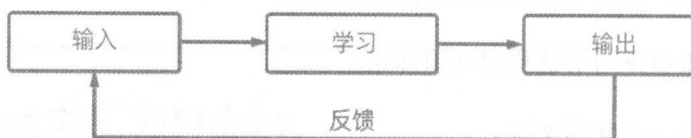


图 11.1 输出是最好的输入

这时就会依据输出来对输入进行反馈，以学习到更多的内容。对团队来说，他们就需要敏捷回顾这样的会议——根据输出的软件、团队协作等内容来改进团队。

11.1.1 增长：回顾与改变

在敏捷团队里，敏捷回顾（retrospect，又称为迭代回顾会议）通常会发生一个迭代的结束与下一个迭代的开始之间，这就有点除旧迎新的感觉。敏捷回顾的目的是激励团队，并促使团队做出改进。其模式的特点是让我们更关注于 Less Well（即实践得不好的地方）。当团队发生了一些不应该发生的事情时，我们就可以在回顾会议上找到原因，并做出相应

的改进，避免发生破窗效应¹。

敏捷回顾是以整个团队为核心去考虑问题的，同时它有一个最高指导原则，即：

无论我们发现了什么，考虑到当时的已知情况、个人的技术水平和能力、可用的资源，以及手上的状况，我们理解并坚信每个人对自己的工作都已全力以赴。

简单地说，就是对事不对人。下面我们来看看在一个团队里是如何进行敏捷回顾的。它不仅可以帮助我们发现团队里的问题，还可以集思广益地寻找一些合适的解决方案。

1. 敏捷回顾的过程

在开始介绍敏捷回顾之前，我们需要准备好一个白板，并在上面写好四个维度，如图 11-2 所示。

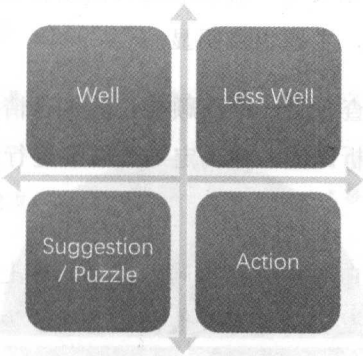


图 11-2 敏捷回顾的四个维度

①好的（Well）。我们在 Well 里记录一些让我们开心的事，如最近天气好、迭代及时完成、没有加班等，这些事从理论上说应该继续保持下去。

1 以一幢有少许破窗的建筑为例，如果那些窗不被修理好，可能将会有破坏者破坏更多的窗户。最终他们甚至会闯入建筑内，如果发现无人居住，也许就在那里定居或者纵火。又或想像一条人行道有些许纸屑，如果无人清理，不久后就会有更多的垃圾，最终人们会视为理所当然地将垃圾顺手丢弃在地上。因此，破窗理论强调着力打击轻微罪行有助于减少更严重的罪案，应该以零容忍的态度面对罪案。

②不好的（Less Well）。关注于在这个迭代的过程中发生了一些什么不愉快的事。一般来说，我们就会对 Less Well 加以细致讨论，找出问题的根源，并试图找到一个解决方案。换句话说，就是要改变现状。

③建议 / 困惑（Suggestion / Puzzle）。如果我们可以直接找到一些建议，那么可以直接提出来，并且如果我们对当前团队里的一些事情有一些困惑，也应该及早提出来。

④实践（Action）。当我们对一些事情有定论的时候，我们就会提出相应的 Action。这些 Action 应该有相应的人去执行，并且由团队来追踪。

整个敏捷回顾的过程如下。

①设定会议目标。在会议开始之前，我们就应该对会议的内容达成一种共识：回顾的主题是啥？要回顾哪些内容？如果是一般性的迭代回顾会议，那么主题就很明显了；如果是针对某一个特定项目的回顾，主题也很明显。

②回顾上一次回顾会议。查看上一个回顾会议的执行情况，检查应该完成的内容是否完成？如果没有完成，应该分析原因，并决定是否继续进行这个事项，或者由其他人继续进行。

③收集数据。我们将会在白板上依据不同的维度，贴上自己的想法、观点、建议等。应该尽可能地保证每个人都有发言，并保证他们都有机会说出自己的想法。

④决定话题。我们会将列表上的所有事项都过一遍，并做一个简单的归类。话题比较少的时候，就可以一个个地过完这些话题；话题比较多时候，就会由团队来投票选择话题讨论。

⑤展开讨论。这种模式的回顾主要关注于 Less Well，因此，我们很容易就可以发现问题。随后展开对问题的剖析，由一些相关的人来描述事情的过程，并展开对发生的过程的讨论，这时候我们仍然是对事不对人——每个人都会犯错，特别是新人，这时就可以考虑在下次遇到这种情况时让有经验的人帮助他。

⑥决定去做一些改进（Action）。我们已经完成了基本步骤，但是最重要的是：决定去做点什么——这传递了一个积极的信息，可以让每个人都有一种感受：我们正在改变现状。

⑦总结与收尾。记录这次回顾的四个维度的内容，并准备好 Action 的内容，以便进行以下的回顾会议。

在这个过程中，提出不好的地方以及做出改进是最重要的。提出不好的内容可以让我们更早地发现问题所在；做出改进便可以改善现状。每一个 Action 都需要有相应的人来执行，即他对这个改进负责，这也意味着这个执行人应该要有能力来改变它。我们不可能在一个迭代里将事情一下变好，但是它可以慢慢地变好。

当我们不断地迭代开发产品，不断地敏捷回顾会议时，我们就会适应这种不断改变来增强自己的方式。在我们习惯了回顾之后，也可以将这项技能带入到日常中——不一定做到“吾日三省吾身”，只需要多省就可以。

那么，接下来让我们对 Growth Studio 项目进行一个简单的回顾。

2. Growth Studio 项目回顾

如果让我对这本书的项目进行一个简单的回顾，那么它一定是如图 11-3 所示的情况。

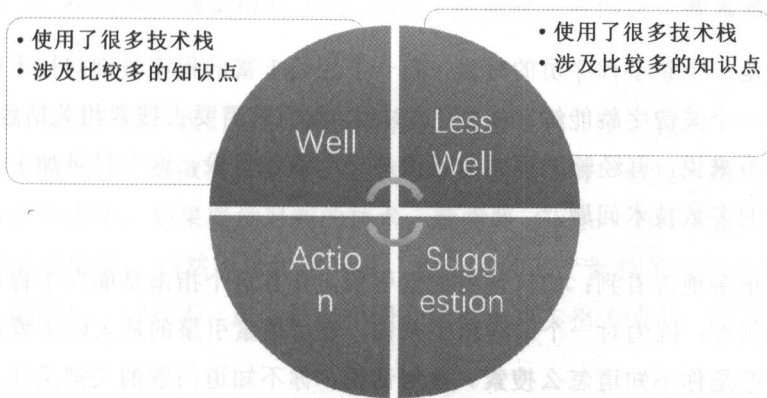


图 11-3 Growth Studio 回顾

在图 11-3 中，Well 和 Less Well 的内容是一样的：

- 使用很多技术栈。
- 涉及比较多的知识点。

这一点也特别有意思，在一个团队里很容易出现这样一种情况：你觉得 A 是一件好事，那么可能就会有 B 觉得这是一件坏事。而对于本书来说，这也是一个事实：涉及的知识点比较广泛，并且在很多点上都比较深入。不同的人会有不同的体验，有的会觉得这是一种挑战；有的人则会觉得像一座大山压在前面；有的可能就会因此而放弃。但是如果能阅读到这里的人，一定属于前者。

下一步应该将回顾的原则应用到个人的学习中。

11.1.2 增长：技能学习与构建索引

在平时学习技术的时候，我们也需要一个类似的反馈过程，只是这种形式并不会那么直观。当我们在某一个项目上使用某个技术的时候，遇到我们不知道的内容，我们就会去搜索相关的知识点。久而久之，如果我们只学需要用到的知识，那么对这个技术的理解就不会全面。这时就应该花些时间去了解这个技术的相关文档，看看我们是否缺失了某些内容。

当我们还是一个新手程序员的时候，向一个经验丰富的程序员询问技术问题时，他只要向我们提供一个关键字就能解决问题。这时候，我们只需要去搜索相关话题的内容即可。对于新手程序员来说，有经验的程序员就像是一个索引目录，这个目录加上搜索引擎，就可以解决遇到的多数技术问题。

你可能在很多地方看到：如何使用搜索引擎，并且这个指南是面向手程序员的。但是这并不能解决问题，因为对一个编程新手来说，使用搜索引擎的最大障碍就是——**你知道哪里有问题，但是你不知道怎么搜索**。换句话说，你不知道问题的关键是什么，问题的关键词又是什么。对新手程序员来说，他们不知道编程世界应该是怎样的，这个地方报错的原因是因为什么。

当我们在学第一门计算机语言时，我们不知道去寻找什么，我们也不知道一些复杂的概念。这时候我们只能看一本别人推荐的书籍，读一读别人留下的笔记，试着去运行一行行代码，借此来一点点了解编程的世界。

而在学习第二门计算机语言的时候，我们有了更多的诀窍——我们开始知道怎么去搜索。在我们的知识体系里，知道如何去搜索，知道编程语言的一些基本特性，就可以通过搜索引擎搜索相关资料进行学习。当然，在这个过程中，最好的资料就是官方提供的文档。

在技术领域，我们与搜索引擎是相似的。对于一个搜索引擎而言，它存在四部分内容：搜索器、索引器、检索器和用户接口。

①搜索器：在互联网中漫游，发现和搜集信息。

②索引器：理解搜索器所搜索到的信息，从中抽取出索引项，用于表示文档以及生成文档库的索引表。

③检索器：根据用户的查询在索引库中快速检索文档，进行相关评价，对将要输出的结果排序，并能按用户的查询需求合理反馈信息。

④用户接口：接纳用户查询、显示查询结果、提供个性化查询项。

对我们而言，我们本身就是用户接口，因此，还需要完善的就是**搜索器 + 索引器 + 检索器**。

搜索器。对我们来说，我们可以通过不同的渠道去了解、学习技术——从书上、博客里了解一些技术的使用，从微博、新闻、公众号中了解最新的一些技术趋势。不同的渠道都有不同的特点和优势：如果要学习新的技术，就需要 GitHub Trending、微博等工具，它们紧紧地跟随着潮流。当然，这也意味着，你学到的某个新技术可能会很快消失。如果要踏踏实实地学习一门技术，那么最简单的方法就是找本相关的书，再深入阅读相关的代码。

索引器。要成为一名可以完成大部分工作的程序员，就需要不断地更新你的索引。我们要学习 Web 开发，就需要对整个 Web 知识体系有一个基本的理解。在不断加深理解的过程中，我们就不断添加了新的资料，然后重新构建索引。

这时我们可以借助思维导图等工具绘制出我们对这个领域的理解。图 11-4 是笔者在之前整理的“嵌入式知识总汇”，在这上面的一个个条目就是索引部分。

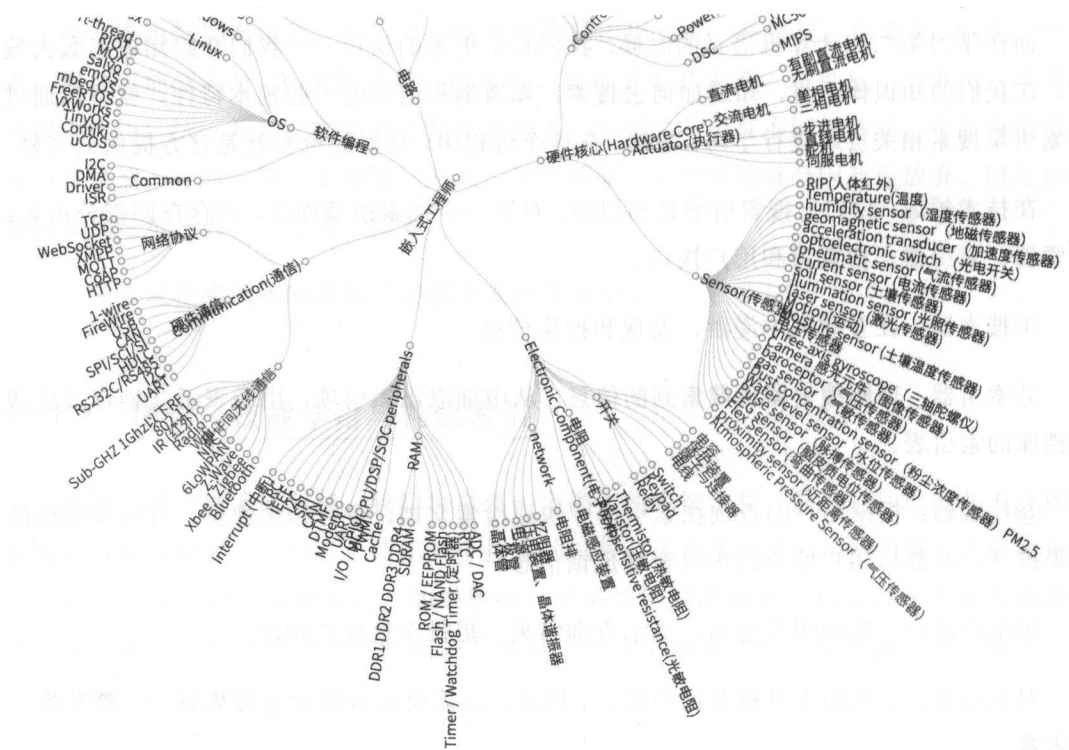


图 11-4 嵌入式知识总汇

我们对整个领域的理解都是由少到多的过程，也是一个持续不断地从输入到输出的过程。

检索器。要完成检索器的相关功能，就意味着：当我们听到某一个技术栈时，我们就有能力将相关的技术罗列出来；同时，我们还能按照需要来组合它们，并设计出一个新的系统。我们也能按照需要设计出几个架构不同的系统，并从中选择最合适的一个。

从搜索器到索引器就意味着，我们开始将输入的知识转换为自己的知识，并且将之沉淀下来。当我们成为一个高级工程师时，我们就需要构建好这个用户接口和索引器，即能从初级工程师的口中分析出他们当前遇到的问题。

在本书里，我们主要侧重于第二点的知识，将我们之前学习到的知识点归类，并将这些知识点一个个串在一起。随后，我们需要做的就是做更多的搜索，不断地更新索引。当

我们不断地去沉淀知识的时候，就可以往更高层级的系统设计迈进。

11.2 设计新架构

如果已经使用了上面的重构、回顾等手段仍然无法拯救现有的系统，就可以考虑愉快地去设计一个新的系统了。但是我们要怎么才能开始呢？

事实上，在前面的章节里已经介绍了设计新架构所需要的基础知识。

- 任务切分，即将目标切换成一个个小的任务，而这些任务尽可能地遵循 SMART 原则，如这里的 18 个步骤。
- 环境搭建，搭建开发所需要的基本环境，并尽最大能力地去练习 IDE 及操作系统等日常工具的使用。
- UI 原型，可以简单地使用 UI 工具来创建 Web 页面的原型。
- 技术选型，能根据我们的能力以及项目的需要选择合适的工具。
- “hello, world”模板，能自己搭建或者寻求一些比较好的应用模板。
- 构建流，可以设计出整个应用的构建流程，如依赖包管理、运行服务、执行语法检测、运行测试等。
- 编码，至少能使用两门以上的语言，一门是 JavaScript，另一门是编译语言。只有动态语言是无法让你理解计算机语言的。
- 测试，可以编写单元测试、服务测试和 UI 自动化测试。
- 部署，可以完整地将应用部署到服务器上。
- 自动化部署，要会在本地输入命令，自动部署新版本的应用到服务器上。
- 分析，不仅要知道很多用户喜欢这个功能，还要知道为什么。

- 优化，既要知道程序中哪里用得更多，又要能对代码进行优化。
- 持续集成，除了学会使用持续集成工具，还要学习什么是持续集成。
- 持续交付，就是缺少一个在持续集成与自动化部署之间的按钮。
- 持续部署，将持续交付的按钮变成自动化就完了？你要改变的不是代码本身，还有组织架构。
- 回顾，由输出结果来改善输入流程，这才能提高组织和自身的水平。
- 重构，使用重构来改进现有代码的设计。

接着，再让我们回到本书的开头，只是用户故事已经变了。我们已经知道：需要一个首页，并且需要一个博客系统和一个面向移动用户的应用。那么，你会怎么设计呢？

每一个程序员都是架构师。平时在我们工作的时候，架构师这个 Title 都被那些非常有经验的开发人员占据着。然而，如果你喜欢刷刷 Github，喜欢做一些有意思的东西，那么你也将会是一个架构师。下面先让我们从设计一个博客系统说起。

1. 搭建一个博客首先应想到的

先问一个问题，如果让你搭建一个博客，你会想到什么技术解决方案？

①静态博客（类似于 GitHub Page）。

②动态博客（可以在线更新，如 WordPress）。

③半动态的静态博客（可以动态更新，但是依赖于后台构建系统）。

④使用第三方博客。

这只是基本的框架。如果只有这点需求，我们将无法规划出整体的方案。现在我们又多了一点需求，我们要求是独立的博客，这样我们就把第 4 个方案去掉。但是就现在的过程来说，我们还是有三个方案。

接着，我们就需要看看：需要什么样的博客？更新频率如何？以及他所能接受的价格？

先说说价格——从价格上说，静态博客是最便宜的，可以使用 AWS S3 或者国内的云存储等。从费用上来说，一个月只需要几元钱，并且快速稳定，可以接受大量的流量访问。而动态博客就贵了很多倍——我们需要一直开着这个服务器，并且如果用户的数量比较大，我们就需要考虑使用缓存。用户数量再增加，我们就需要更多的服务器。而对于半动态的静态博客来说，需要有一个 Hook 检测文章的修改，这样的 Hook 可以是一个客户端。当修改发生的时候，运行服务器，随后生成静态网页。最后，这个网页将部署到静态服务器上。

从操作难度上说，动态博客是最简单的，静态博客紧随其后，半动态的静态博客是最难的。整个性价比考虑如表 11-1 所示。

表 11-1

	动态博客	静态博客	半动态的静态博客
价格	几十到几百元	几元	依赖于更新频率 几元至几十元
难度	容易	稍有难度	难度稍大
运维	不容易	容易	容易
数据存储	数据库	无	基于 git 的数据库

现在，我们已经达成了一定的共识，有了几个方案可以供用户选择。而这时，我们并不了解进一步的需求，只能等下面的结果。

客户需要看到文章的修改变化，这时就去除了静态博客。现在还有第 1 和第 3 种方案可以选，考虑到第 3 种方案实现难度比较大，不易短期内实现，并且第 3 种方案可以依赖于第 1 种方案，就采取了动态博客的方案。

但是，问题实际上才刚刚开始。

2. 使用的技术

作为一个团队，我们需要优先考虑这个问题。使用什么技术解决方案？而这是一个更复杂的问题，这取决于我们团队的技术组成，以及未来的团队组成。

如果在现有的系统中使用的是 Java 语言，并不意味着每个人都喜欢使用 Java 语言。因为随着团队的变动，做这个技术决定的那些人有可能已经不在这个团队里。即使那些人

还在，也并不意味着我们喜欢在未来使用这个语言。当时的技术决策都是在当时的环境下产生的，在现在看来很奇怪的技术决策，有可能在当时是最好的技术决策。

对一个优秀的团队来说，不存在一个人对所有的技术栈都擅长的情况——除非这个团队所从事的业务范围比较小。在一个复杂的系统里，每个人都负责系统相应的一部分。尽管到目前为止并没有好的机会去构建自己的团队，但是也希望总有一天有这样的机会。在这样的团队里，只需要有一个人负责整个系统的架构。其余的人可以在自己擅长的层级里构建自己的架构。因此，让我们再回到博客中，现在我们已经决定使用动态的博客。然后呢？

作为一个博客，我们至少要有前后台，这样可能就需要两个开发人员，如图 11-5 所示。

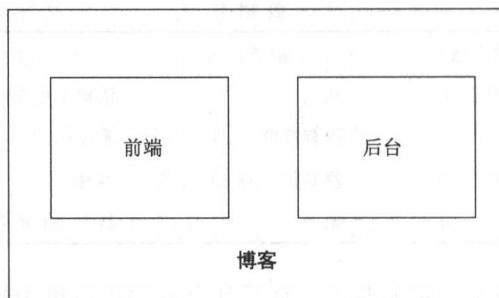


图 11-5 前后台

当然，我们也可以使用 React，但是在这里先让我们忽略这个框架，紧耦合会削弱系统的健壮性。接着，作为一个前端开发人员，还需要考虑以下两个问题。

- 我们的博客系统是否是单页面应用？
- 要不要做成响应式设计。

第二个问题不需要和后台开发人员做沟通就可以做决定了。而第一个问题，我们则需要和后台开发人员做决定。单页面应用的天然优势就是：由于系统本身是解耦的，他与后台模板系统脱离。这样在我们更换前端或者后台的时候，不需要去考虑使用何种技术——因为我们使用 API 作为接口。现在，我们决定做成单页面应用，那么就需要定义一个 API。之后，我们就可以决定在前台使用何种框架：Angular、Backbone、Vue、jQuery，接着我

们的架构可以进一步完善，如图 11-6 所示。

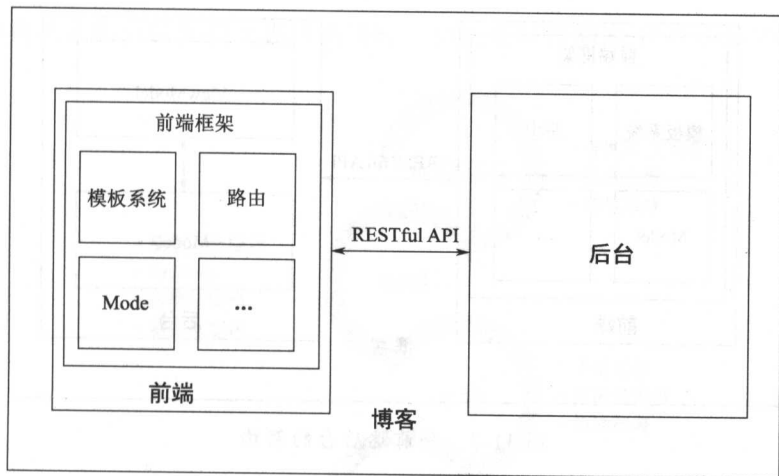


图 11-6 含前端的架构

这时，后台人员也可以自由地选择自己的框架、语言。后台开发人员只需要关注于生成一个 RESTful API 即可，而他也需要一个好的 Model 层来与数据库交付。

现在，我们似乎已经完成了大部分工作。除此之外，我们还需要做以下工作。

①部署到何处操作系统。

②使用何处的数据库。

③如何部署。

④如何分析数据。

⑤如何测试。

我们需要按之前介绍的步骤，一步步往下执行，最后才能完成整个系统的设计，如图 11-7 所示。

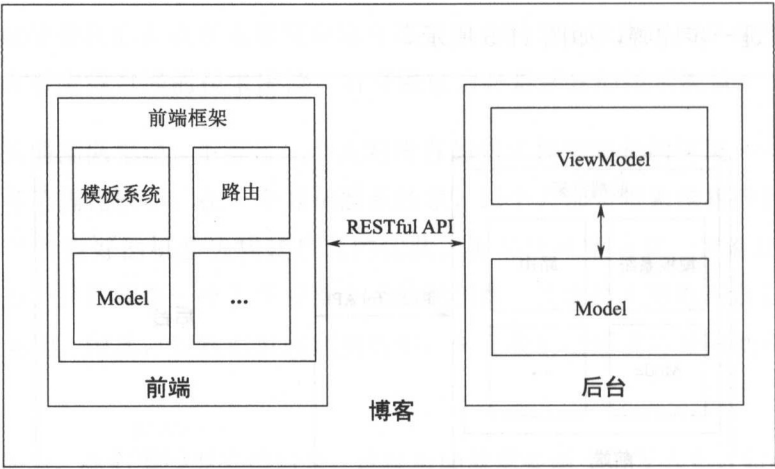


图 11-7 含前端后台的架构

3. 架构与模式

对于代码设计来说，使用设计模式来抽象化代码是一种简便的设计方式。对于架构设计来说，最简单的设计方式是使用现有的模式。我们在第 3 章中提到的 MVC 架构是一种常用的分层架构模式，除此之外，还有诸如管道和过滤器模式等常见的架构模式。

这些模式不是一开始就有的，好的软件也不是一开始就设计成现在这样的，好的设计亦是如此。导致我们重构现有系统的原因有很多，但是多数是因为原来的代码变得越来越不可读，并且重构的风险太大了。在实现业务逻辑的时候，我们快速地用代码实现，没有测试，没有好的设计。

在需求变化的过程中，一个设计的模式本身也是在不断地改变。如果我们还固执于原有的模式，就会犯下一个又一个的错误。

在适当的时候改变原有的模式，进行一些演进显得更有意义一些。如果我们不能在适当的时候引进一些新的技术，那么旧有的技术就会不断累积。这些技术债就会不断往下叠加，从而导致这个系统接近崩塌。而我们在一开始所设定的一些业务逻辑也会随着系统逝去，这个公司似乎也要到尽头了。

如果我们可以不断地演进系统——抽象服务、拆分模块等。业务在技术不断演进的过

程中将保留下来，而软件架构本身将会不断地更新。

当我们再次使用新的架构来设计系统时，仍然要经过如图 11-8 所示的过程。

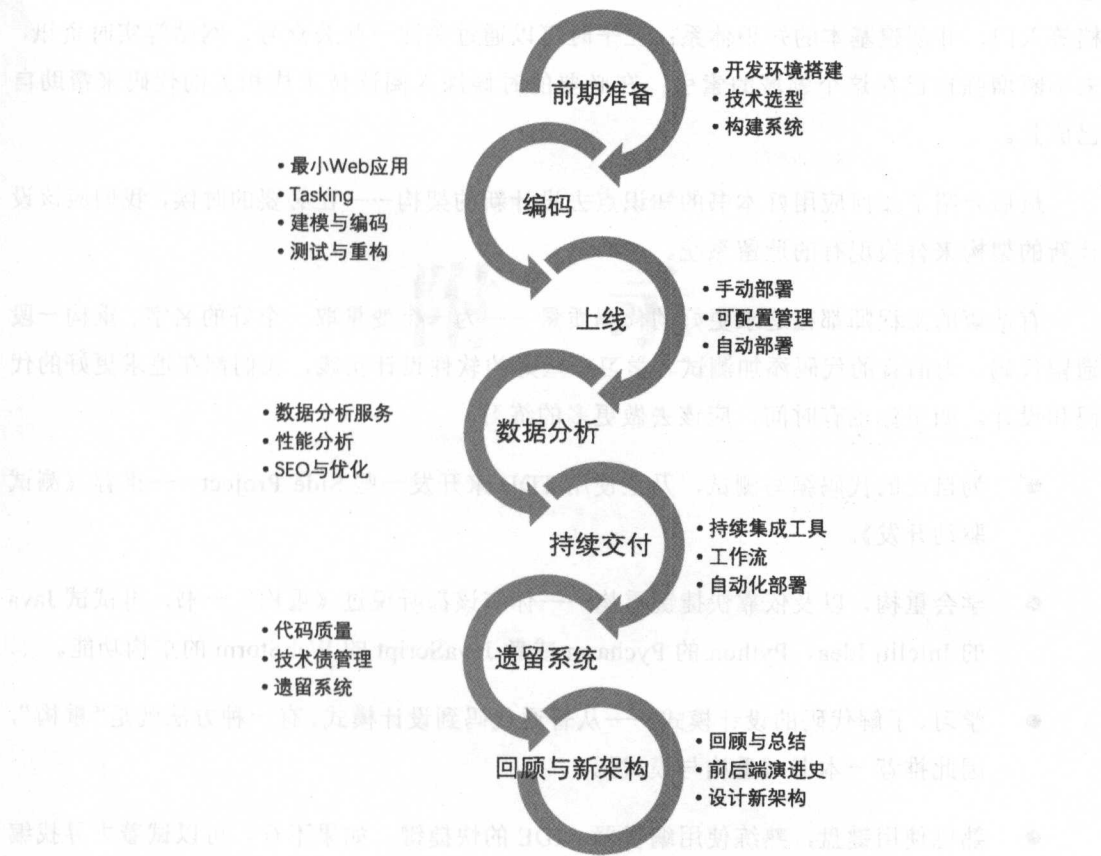


图 11-8 Web 应用的生命周期

这就好像是 Web 应用的模式一样，我们一直在这样一个循环里。

11.3 小结

在本书的最后一章里，我们介绍了如何使用敏捷回顾的技巧来持续性地对团队进行改

进。将回顾的技巧运用到项目上，不仅可以帮助改进团队，还能及时帮我们发现软件开发中的问题。经常对自己在过去的某一周期内进行回顾，也能帮助我们发现成长过程中遇到的问题。随后，我们对如何渐进性地学习技术进行了介绍，通过书籍、博客、官方文档等入门，并创建基本的知识体系；在平时可以通过关注一些公众号、网站等实时资讯，来不断增强自己在这个领域的索引；在必要的时候深入阅读技术栈相关的代码来帮助自己成长。

最后介绍了如何应用好本书的知识点去设计新的架构——在必要的时候，我们应该设计新的架构来替换现有的遗留系统。

有洁癖的工程师都在追求更好的代码质量——为一个变量取一个好的名字、重构一段遗留代码、为旧有的代码添加测试、学习一些好的软件设计实践，我们都在追求更好的代码和设计。如果你也有时间，应该去做更多的练习。

- 为自己的代码编写测试，乃至使用 TDD 来开发一些 Side Project——推荐《测试驱动开发》。
- 学会重构，以及依靠快捷键重构——你应该都听说过《重构》一书，再试试 Java 的 IntelliJ Idea、Python 的 Pycharm 或者 JavaScript 的 WebStorm 的重构功能。
- 学习、了解代码的设计模式——从普通代码到设计模式，有一种方法就是“重构”，因此推荐一本书《重构与模式》。
- 熟练使用键盘，熟练使用编辑器、IDE 的快捷键。如果不行，可以试着去寻找编辑器和 IDE 的 cheatsheet，像 VIM 和 Emacs 这样的编辑器，你就可以买相应的书参考。
- 尽可能多地探索提高效率的工具。这一点也可以参考我的 Toolbox (<https://github.com/phodal/toolbox>) 建立自己的高效工具箱。
- 有机会尽可能去尝试新的技术。建议可以看看“ThoughtWorks 技术雷达”，你可以了解到一些技术趋势。

1. 一点点真实学习体验

在开始学习一门编程语言或新技术的时候，我们可能会从“hello world”开始。

好了，现在来谈 Scala 语言的初学者，使用搜索引擎搜索 [Scala] 来看看 [Scala] 是什么：

Scala 是一门类 Java 的编程语言，它综合了面向对象编程和函数式编程。

接着又开始看 [Scala tutorial]，这时找到了第一个示例：

```
object HelloWorld {
  def main(args: Array[String]) {
    println("Hello, world!")
  }
}
```

附录

A 类型

获取到 5% 的知识。

这门语言与 Java 语言相似，其
目录如下。

朱姓的匿名学回咬

● 表达式和值。

● 函数是一等公民。

● 模式匹配。

● 按名称传递参数。

● 定义类。

● 单子类型。

● 抽象化。

● 类型。

附录 A

如何学习新的技术

1. 一次语言学习体验

在开始学习一门语言或者技术的时候，我们可能会从“hello, world”开始。

好了，现在我是 Scala 语言的初学者，我用搜索引擎搜索『Scala』来看看『Scala』是什么：

Scala 是一门类 Java 的编程语言，它结合了面向对象编程和函数式编程。

接着又开始看『Scala 'hello,world'』，然后找到这样一个示例：

```
object HelloWorld {  
  def main(args: Array[String]): Unit = {  
    println("Hello, world!")  
  }  
}
```

获取到 5% 的知识。

这门语言与 Java 语言相比看上去还行。然后找到一本名为『Scala 指南』的电子书，目录如下。

- 表达式和值。
- 函数是一等公民。
- 借贷模式。
- 按名称传递参数。
- 定义类。
- 鸭子类型。
- 柯里化。
- 范型。

- Traits。

看上去还行，又得到了 5% 的知识点。接着，依照上面的代码和搭建指南在自己的电脑上安装了 Scala 的环境：

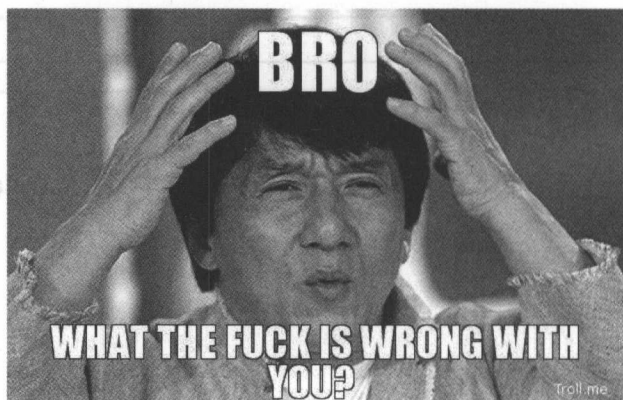
```
brew install scala
```

Windows 用户可以用：

```
choco install scala
```

然后开始写一个又一个的 Demo，感觉自己得到了很多特别的知识点。

到了第二天忘了！

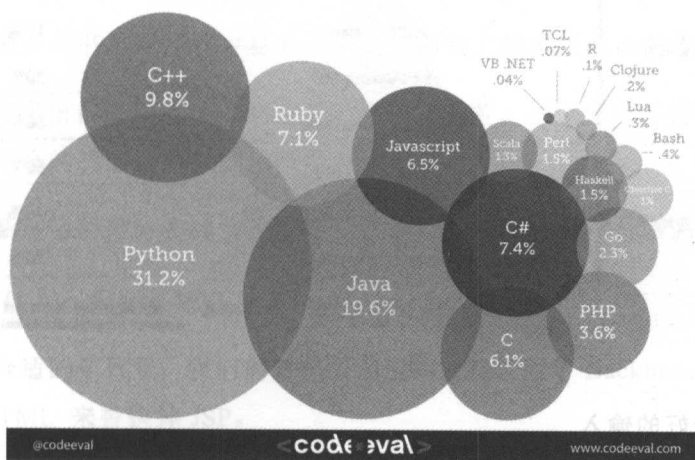


接着，你又重新把昨天的知识过了一遍，还是没有多大的作用。突然间，你听到别人在讨论什么是这个世界上最好的语言——你开始加入讨论。

于是，你说出了 Scala 这门语言可以：

- 支持高阶函数，lambda，闭包。
- 支持偏函数，match。
- mixin，依赖注入。

Most Popular Coding Languages of 2015



最流行的语言

你发现隔壁的 Python 小哥之所以赢得了这场辩论，是因为他把 Python 语言用到了各个地方——机器学习、人工智能、硬件、Web 开发、移动应用等。而你还没有用 Scala 写过一个真正的应用。

让我想想我能做什么？我有一个博客。对，我有一个博客，可以用 Scala 把我的博客重写一遍：

①先找一个 Scala 的 Web 框架，Play 看上去很不错，就这个了。这是一个 MVC 框架，原来用的 Express 也是一个 MVC 框架。Router 写这里，Controller 类似这个，就是这样的。

②既然已经有 PyJS，也会有 Scala-js，前端就用这个。

好了，博客重写了一遍，感觉还挺不错的，我决定向隔壁的 Java 小弟推销这门语言，以解救他于火海之中。

『让我想想我有什么杀手锏？』

『这里的知识好像还缺了一点，这个是什么？』

好了，你已经得到了 90% 的知识，如图 A-1 所示。希望你能从这张图中得到很多点。



图 A-1 Learn

2. 输出是最好的输入

图 A-2『学习金字塔』就是在说明“输出是最好的输入”。

如果你不试着去写点博客、整理资料、准备分享，那么可能并没有意识到你缺少了多少东西。

因为你一直在完善功能、完成工作，你总会有意、无意地漏掉一些知识，而你也没有意识到这些知识的重要性。

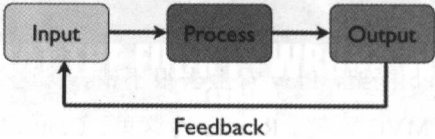


图 A-2 Output is Input

从我有限的（500+）博客写作经验里，我发现多数时候需要更多地参考资料才能更好地向他人展示这个过程。为了输出，我们需要更多的输入，进而加速这个过程。

如果是写书，则是一个更高水平的学习，你需要发现别人在他们的书中欠缺的一些知识点，并且还要展示一些在其他书中没有，而这本书会展现这个点的知识，这意味着你需要挖掘得更深。

所以，如果下次有人问你如何学一门新语言、技术，那么答案就是写一本书。

3. 应用新的技术

对大多数人来说，写书不是一件容易的事，而应用新的技术则是一件迫在眉睫的事。

通常来说，技术出自对现有技术的改进。这就意味着，在掌握现有技术的情况下，我们只需要做一些小小的改动就可以实现技术升级。

而学习一门新技术的最好实践就是用这门技术对现有的系统进行重写。

第一个系统 (v1): Spring MVC + Bootstrap + jQuery

那么在那个合适的年代里，我们需要单页面应用，就使用了 Backbone。然后，就可以用 Mustache + HTML 来替换掉 JSP。

第二个系统 (v2): Spring MVC + Backbone + Mustache

这时我们已经实现了前后端分离，系统实现上变成了这样。

第二个系统 (v2.2): RESTful Services + Backbone + Mustache

或者

第二个系统 (v2.2): RESTful Services + AngularJS 1.x

Spring 只是一个 RESTful 服务，我们还需要一些问题，比如 DOM 的渲染速度太慢了。

第三个系统 (v3): RESTful Services + React

系统就是这样一步步演进过来的。

尽管系统最后的架构已经不是当初的架构，而系统本身的业务逻辑变化并没有发生太大变化。

特别是对如博客这类系统来说，其技术实现已经趋于稳定，而且是你经常使用的东西。所以，下次试试用新技术的时候，可以先从你的博客开始。

附录 B

安装 Piwik

(1) 安装 PHP 模块

```
$ sudo apt-get install php-mysql php-curl php-gd php-intl php-cli php-geoip  
php-fpm
```

(2) 获取最新的 Piwik 源码

```
$ cd /var/www/html/  
$ sudo wget http://builds.piwik.org/latest.zip  
$ sudo unzip latest.zip
```

(3) 为源码设置权限

```
$ sudo chown -R www-data:www-data /var/www/html/piwik  
$ sudo chmod -R 0755 /var/www/html/piwik/tmp
```

(4) 安装数据库

```
sudo apt install mariadb-server
```

(5) 创建数据库

```
sudo mysql -u root -p  
MariaDB [(none)]> CREATE DATABASE piwikdb;  
MariaDB [(none)]> CREATE USER piwik@localhost IDENTIFIED BY 'piwik';  
MariaDB [(none)]> GRANT ALL PRIVILEGES ON piwikdb.* TO piwik@localhost;  
MariaDB [(none)]> FLUSH PRIVILEGES;  
MariaDB [(none)]> exit
```

(6) 启动 PHP 服务

```
systemctl start php7.0-fpm
```

(7) 添加 Nginx 配置

```
server {
```

```
listen 8088;

# Parameterization using hostname of access and log filenames.
access_log /var/log/nginx/piwik_access.log;
error_log /var/log/nginx/piwik_error.log;

# Disable all methods besides HEAD, GET and POST.
if ($request_method !~ ^(GET|HEAD|POST)$ ) {
    return 444;
}

root /var/www/html/piwik/;
index index.php;

# Disallow any usage of piwik assets if referer is non valid.
location ~* ^.+\.(\?:jpg|png|css|gif|jpeg|js|swf)$ {
    # Defining the valid referers.
    valid_referers none blocked *.mysite.com othersite.com;
    if ($invalid_referer) {
        return 444;
    }
    expires max;
    break;
}

# Support for favicon. Return a 204 (No Content) if the favicon
# doesn't exist.
location = /favicon.ico {
    try_files /favicon.ico =204;
}

# Try all locations and relay to index.php as a fallback.
location / {
    try_files $uri /index.php;
```

```

}

# Relay all index.php requests to fastcgi.
#location ~* ^/(?:index|piwik)\.php$ {
#    fastcgi_pass UNIX:/run/php/php7.0-fpm.sock;
#}

location ~ \.php$ {
    try_files $uri =404;
    fastcgi_split_path_info ^(.+\.(php))(/.+)$;
    fastcgi_pass UNIX:/var/run/php/php7.0-fpm.sock;
    fastcgi_index index.php;
    fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
    include fastcgi_params;
}

# Any other attempt to access PHP files returns a 404.
location ~* ^.+\.php$ {
    return 404;
}

# Return a 404 for all text files.
location ~* ^/(?:README|LICENSE[^\.]*|LEGALNOTICE)(?:\.(txt))*$ {
    return 404;
}
}

```

(8) 让应用运行

```
sudo ln -s /etc/nginx/sites-available/piwik /etc/nginx/sites-enabled
```

下一步进入数据库设置，添加之前创建的数据库，填入账号（不包含 @localhost）和密码。建立超级用户后，就可以使用了。



欢迎反馈意见或投稿
邮箱: dongying@phei.com.cn
电话: 010-88254047
微信号: yingzidd

作者基于大量实践凝练而成的全栈工程师技能图谱，对任何想成为全栈的前端或后端开发人员来说，本书都能给你带来帮助。如果我早点结识这本秘籍，也许我早就是一名优秀的架构师了。

——百度高级前端工程师 颜海镜

所谓全栈工程师，竞争力是什么呢？首先，能够从全局上把握项目的开展，对整体开发链路和技术体系有深入的理解；其次，就是掌握更多的知识（点）和工具，在每个环节都比别人了解得更全面、细致。Phodal的这本书恰好可以给你带来这些能力。

本书围绕Web全栈开发涉及的整体循环链路做了充分说明，时而穿插生动、详细的实战过程。看一遍，你可以更加深入地理解全栈工作；若能跟着实操，必会受益匪浅。书中涉及大量日常工作中不可错过的知识点详解，学一个赚到一个，值得推荐。

——资深 Web 研发工程师 小胡子哥

有幸作为早期预览者看到作者以非常严谨的态度在写这本书。这本书的问世是众多开发者的福音，作者以简单易懂、风趣的笔风为大家掀开了作为全栈工程师应该学习和具备的能力。本书将能让大家更好、更清楚地理解全栈工程师的成长历程，同时也可以帮助技术团队解决很多问题，提高开发效率，降低开发成本，也许会成为一些团队提高工程师能力的一种契机。

——W3cplus.com站长 大漠

本书通过大量的实例深入浅出地讲解了全栈开发的最佳实践，对基础概念的讲解也是抽丝剥茧、鞭辟入里，使枯燥的知识顿时鲜活起来。作者三年磨一剑，不管你是前端小白，还是全栈专家，本书都值得一看。

——在线回声前端技术专家 justjavac

全栈意味着我们不介意跳出自己的专业思考和解决问题；而精益则鼓励我们接近问题的本质和真相，然后大胆取舍，抓住那些四两拨千斤、以不变应万变的东西。这两方面的思考在当今社会中都特别珍贵。这本书可以帮助你了解很多被广泛实践的方法，以及被广泛应验的“套路”，也从个人和团队的角度谈了很多。如果你是全栈或精益的信徒，不妨读一读这本书。

——阿里巴巴技术专家 赵锦江（花名：勾股）



博文视点Broadview



@博文视点Broadview



策划编辑：董 英
责任编辑：李利健
封面设计：李 玲

上架建议：Web开发

ISBN 978-7-121-31369-1



9 787121 313691 >

定价：79.00元